Jül-4345

Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Development of a Multiple View Visualisation Framework for Large Particle Sets

A. M. Westhoff

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Development of a Multiple View Visualisation Framework for Large Particle Sets

A. M. Westhoff

Berichte des Forschungszentrums Jülich; 4345
ISSN 0944-2952
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)
Jül-4345

Vollständig frei verfügbar im Internet auf dem Jülicher Open Access Server (JUWEL)
unter http://www.fz-juelich.de/zb/juwel

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek, Verlag
D-52425 Jülich · Bundesrepublik Deutschland
Z 02461 61-5220 · Telefax: 02461 61-6103 · e-mail: zb-publikation@fz-juelich.de

Abstract

This thesis deals with the development of a visualisation framework specialised for large particle sets which have been simulated on supercomputers. It provides a modular design so that multiple input formats are supported. Furthermore, multiple output devices are adaptable ranging from laptops over desktop computers to stereoscopic projection systems.

To come up against the large size of input data with thousands of time steps containing up to billions of particles, a concept of multiple views has been established. Each view consists of an operator choosing a subset of the input data and a filter defining the graphical interpretation of this subset. The filters describe three-dimensional scenes or two-dimensional displays. The scenes are inserted in the main window directly, all views containing two-dimensional filters are adapted to the border of the main window as overlays.

Zusammenfassung

In dieser Arbeit wird die Entwicklung eines Visualisierungs-Frameworks für große, auf Supercomputern simulierte Partikel-Datensätze vorgestellt. Es bietet eine modulare Struktur und kann um verschiedene Eingabeformate erweitert werden. Die Ausgabe kann auf unterschiedlichen Geräten erfolgen, von Laptops über Desktop-Systeme bis hin zu stereoskopen Projektionsflächen.

Um die enorme Anzahl der Eingabedaten mit tausenden von Zeitschritten und jeweils bis zu mehreren Milliarden von Partikeln verarbeiten zu können, wird ein Konzept mit multiplen Views eingesetzt. Jede dieser Views besteht aus einem Operator, der eine Teilmenge der Eingabedaten auswählt, und einem Filter, der die graphische Darstellung dieser Menge definiert. Die Filter können sowohl dreidimensionale Szenen als auch zweidimensionale Anzeigeformen beschreiben. Die szenischen Interpretationen werden alle in das gleiche Fenster integriert. Die Views, die andere Filtertypen beinhalten, werden als Overlays entlang der Ränder in diesem Fenster angeordnet.

Contents

List of Figures

List of Tables

Listings

1 Motivation

Jülich Supercomputing Centre of Forschungszentrum Jülich GmbH operates two of the fastest supercomputers worldwide. One of them, called "JUGENE" as an abbreviation for "Jülicher Blue Gene/P", was on the ninth place in the TOP500 list from November 2010 and is still on the twelfth place in this list published in June 2011[1]. These supercomputers are heavily used for simulations, mainly from the field of physics.

These physical simulations computed on thousands of CPU nodes generate huge amounts of output data. Often these results have to be processed for publications or talks. In most cases the output data contains much more information than needed for these purposes because only some of the parameters are of interest. To give an example, a lot of different physical properties have been computed but only a graph of the charge distribution is required. Thus a visualisation is needed that can extract the subsets to be visualised automatically via configurations so that the user does not have to search through the data for himself to extract the data subsets.

Dealing with simulations on supercomputers, different scenarios exist concerning the way to transfer the output of the calculations as input data to the visualisation. The data can be directly streamed from the compute nodes to the visualising computer to get a live visualisation. Another possibility is to save the simulated data into an external storage and read them in later for the visualisation. A visualisation framework shall provide interfaces for both scenarios.

Also concerning the output media of the visualisation, there are a lot of variants. On the one hand, there are workstations, stand-alone computer and notebooks, on the other hand, there are high-end graphical systems with stereoscopic projections. The possibility to change between the different output media without any additional effort is advantageous for the user.

To conclude, a modular visualisation framework is needed providing a simple way to select data subsets and drivers for multiple inputs and multiple outputs to get a user-optimised and easy to handle visualisation frontend. This thesis deals with the development of such a visualisation framework.

In the first chapter, the term "Particle Simulation" is introduced because the visualisation has to be specialised to a certain type of data. With the Coulomb Solvers and especially the Fast Multipole Method different well-known algorithms for such simulations are briefly presented. Subsequently the properties significant for physical particle simulations are explained, namely physical properties on the one hand and simulation properties on the other hand.

The following section 2.2 deals with the question why visualisations are needed at all. Therefore, already existing visualisation tools and frameworks like OpenGL, ViSTA and ParaView are evaluated with respect to the three main demands on the

[1] http://www.top500.org/

framework listed above. Out of this, the requirement for the development of a new framework emerges so that the demands on this framework are specified in detail.

Chapter 3 deals with the development of the software model forming the basis of the framework. First, some general aspects like the choice of the programming language are explained. Afterwards the data model is described which distributes the different tasks to several modules. The internal storage model, the interfaces for the inter-module communication and at last the external interfaces for the communication with the underlying frameworks are specified.

The following section 3.3 deals with the development of a multiple view concept. This is comparable to the views used in databases providing distinct and restricted views on the same data set for different user groups so that every operator has only access to those subsets of data it is permitted to use. In case of the visualisation framework, the views are three-dimensional displays in a single scene and two-dimensional plots as window overlays.

For each of these views, like with databases, the data subset to be included in the view has to be defined, so an input selection has to be evaluated. This procedure is done by the so-called "operators" in the framework. Their functionality is explained in section 3.4. Furthermore, it is emphasized how arbitrarily complicated operators can be defined writing XML code and how they are internally stored after a parsing process.

The extracted subset has to be visualised in a further step. There are a lot of possibilities to interpret particle data graphically like scatter-plots, vector fields or (two-dimensional) diagrams. The graphical interpretation is in principle independent of the chosen data subsets and flexible, so that the following section 3.5 deals with the development of the so-called "filters". They form a way to define the different graphical interpretations independent of the data to be displayed specifying the display mode and parameters like the colours to be used.

As the last main part of the framework development, the data storage is described in section 3.6. It deals with the question of how to store the input data internally most adequately for a framework providing a concept with multiple views, operators and filters.

Chapter 4 concludes the main aspects of the implementation of the visualisation framework highlighting what is new in contrast to the already existing frameworks. Furthermore, aspects are denoted which have not worked out or which pose bottlenecks.

This thesis ends with chapter 5 describing which features are missing although they are desired and how they can be added to the framework utilising the modular project concept. An example for these future extensions are parallel input and output or the control during runtime using a tablet computer.

Appendix A contains a detailed dependency graph displaying the relations between the different modules. The following appendix B comprehends examples of all possible XML files needed to configure the program and to define operators, filters and views. Appendix C demonstrates with an example how the input data

can be structured and how this structure has to be defined in the configuration XML files.

The last three appendices D, E and F summarise all information necessary to use the framework, namely the system requirements, the platforms it has been tested on and the license.

2 Introduction

2.1 Particle Simulations

2.1.1 Particle Simulations in General

In order to answer the question what particle simulations are in general, first it has to be defined what a physical particle is. It is a "body which is concentrated in a single point"[5, p. 25]. It has no extension so that it is zero-dimensional. Different properties can be assigned to it like mass or charge. All forces of the particle are concentrated in this point. Different forces may take effect on a particle. These can be outer forces or forces between two particles.[5, p. 25][24]

Particle simulations calculate the changes within a system of particles time-dependently, so how all particle properties change during chronological sequence. Here all influences on a particle are regarded. These can be outer influences and the interactions of all particles with others. So the influence of all particle properties on the ones of all other particles are calculated which is an alternative to the solution of differential equations.

Possible fields of applications are, amongst others, molecular dynamics[9], plasma physics[8], astro physics[25] and fluid dynamics[26]. This wide range of fields demonstrates that particle simulations can be used in every context dealing with bodies whose forces can be seen as concentrated in a single point.

Since the interaction of every particle with each other particle has to be calculated, the intuitive implementation of particle simulations has got the complexity $O(N^2)$ for a particle set with N elements. Many different approaches exist trying to reduce the complexity. A well-known class are the Coulomb solvers for reducing the computational effort to $O(N \log N)$ or even $O(N)$[11, p. 10ff]. So the system size, the number of particles to be used, can be chosen much higher. The approaches of the Coulomb solvers and one of them in particular, the Fast Multipole Method, are introduced in the following section.

2.1.2 Coulomb Solvers – Algorithms for Long Range Potentials and Forces

In computer simulations of particle systems, the evolution of interacting particles is approximated by time using the integration of the equations of motion. Since each particle influences all other, each of the N particles in the system interacts with $N-1$ other particles so that all in all $\frac{1}{2}(N-1) \cdot N = \frac{1}{2}(N^2 - N)$ interactions have to be examined for each time step. So an algorithm of this kind has the complexity $O(N^2)$. For small particle sets the simulation can be implemented in this intuitive way. For large sets the effort is immense, even if a supercomputer is used.

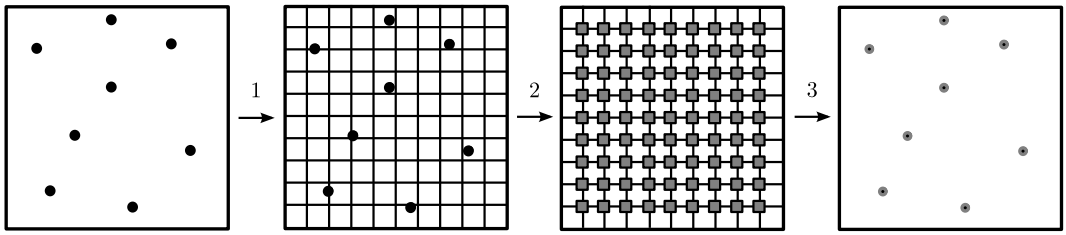


Figure 2.1: This figure illustrates the idea of grid-based algorithms for the calculation of the long range parts of the potential.[11, p. 262]

Therefrom for large particle systems a more efficient algorithm must be used.

These faster algorithms depend a lot on the type of interactions between the particles. For example in case of short range potential functions, only these particles have to be included in the calculation which are close enough to the considered one. The forces are evaluated using the gradient of the potential. In case of long range potentials, like the one used for the evaluation of the Coulomb force, particles cannot be totally pruned even if they are far away from the actually considered particle. Here one does not have to differentiate between two particles close to each other which are far enough away from the considered one since the resulting forces and potentials are nearly the same. This behaviour is used in algorithms for long range potentials. The field in which each particle has to be evaluated individually is called near field, the one where particles can be summed is the far field.

In the following, only these solvers will be described which are dealing with long range forces, for example with the Coulomb force. There are two main approaches for these algorithms, namely grid-based and tree-based algorithms.[11, p. 7-14]

Grid-Based Algorithms

For grid-based algorithms, the total potential is divided into two parts, a short range and a long range one. The short range potential can be evaluated with one of the short range algorithms like the Linked Cell (for more information see [11, p. 59]). The long range potential is once more divided into a short and a long range part. This short range part can as well be calculated with the Linked Cell algorithm. The long range part can be approximated using a grid-based algorithm.

The grid-based algorithms mainly consist of three steps illustrated in figure 2.1.

First, a grid is applied to the domain containing all particles. The finer this grid is meshed, the more accurate is the approximation. In the second step, the charge density induced by the particles is approximated to the grid. In the third step, the potential equation is solved to calculate the potential on all grid points. The result is now interpolated from the grid nodes to the particle positions.

The approximations of this approach do not differ a lot from the exact calculation in case of smooth functions. Otherwise another decomposition in a smooth and

a singular part is needed. The singular part has to be calculated like the short range parts, the smooth part can be approximated using the grid-based algorithm.

The grid-based algorithms work best if the particles are uniformly distributed. For non-uniform particle distributions, in other words if particle clusters exist, they are less efficient. This kind of data often exists in astro physics and molecular dynamics. In such cases, tree-based algorithms are more efficient.[11, p. 247ff]

Tree-Based Algorithms

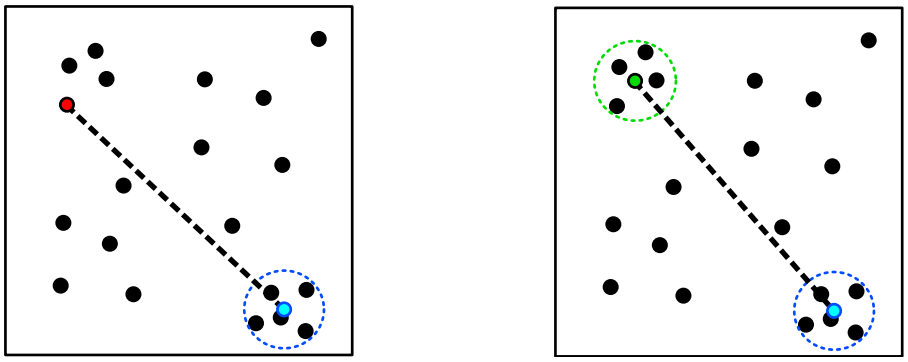
Tree-based algorithms use a hierarchical decomposition of the base domain. They can be parallelised more efficiently than grid-based methods. The density distribution is approximated adaptively for the decomposed domains. Afterwards, the potential is calculated differentiating between near field and far field. The resulting methods have complexities ranging from $O(N \log N)$ to $O(N)$. The Fast Multipole Method is an algorithm achieving $O(N)$ which is described in the following subsection.

All tree-based algorithms use tree structures to administrate the recursive domain decomposition in cells of varying size. The root of the tree represents the base domain. This domain is decomposed in disjoint subdomains which are then assigned to the child nodes of the root node. Now each of these child nodes is decomposed the same way. A node is not decomposed if it contains no or one single particle.

For the domain decomposition, different variants are possible: Each domain can be divided in half. Alternatively the subdomains have different sizes depending on the density distribution so that the particles are distributed equally into the subdomains. For each inner node of the tree, we represent the particles inside this box as a pseudo-particle.

For the parallelisation, all tree nodes, hence all subdomains, have to be distributed to the CPUs whereat a balanced load distribution is important to minimise the compute time of the simulation. So for each level of decomposition, a space-filling curve (see section 2.1.2) of the subdomains is needed.

Now the different depth of the tree nodes can be used to identify the near and far field to simulate the particles. Here the pseudo-particles are used based on the idea that the interactions of a particle with particles in a distant cell is nearly the same as the interaction with the pseudo-particle of this cell. So in this case particle-cluster-interactions are used (Method of Barnes-Hut, see [11, p. 335ff]). The complexity of this method is $O(N \log N)$ because each of the N particles has to interact with $O(\log N)$ pseudo particles.[11, p. 323ff]



(a) Barnes-Hut (b) Fast Multipole Method

Figure 2.2: The two figures compare the *Barnes-Hut* algorithm with the *Fast Multipole Method* to show the difference between particle-cluster-interactions (Barnes-Hut) and cluster-cluster-interactions (FMM)[11, p. 326, 390].

A Fast Tree-Based Solver: The Fast Multipole Method

The Fast Multipole Method (FMM) is a tree-based algorithm like the method of Barnes-Hut. This method achieves the optimal complexity of $O(N)$ by the additional use of cluster-cluster-interactions.

The Barnes-Hut algorithm computes for each of all close particles the interactions with distant pseudo-particles, so particle-cluster-interactions. The main idea of the FMM is also to create clusters of those particles which have been examined individually by Barnes-Hut so that in the end cluster-cluster-interactions can be calculated. The difference between these two approaches is illustrated in figure 2.2. After computing the cluster-cluster-interactions, the result represents several particle-particle-interactions between the two clusters.

In the FMM algorithm, the particle data is sorted in the leaf nodes of the tree. The node on the parent level of the leaves contains the first multipoles or pseudo-particles which are directly computed out of the particles in the leaves. These multipoles are expansions of all particles within the defined area in a single point. On all higher node levels, the multipoles are computed using the multipoles of the child nodes. The multipole-moments of higher levels are computed on each level of the tree beginning with the level of the child nodes of the root node.

In the end, the interactions for every pair of particles has been computed. Figure 2.3 illustrates for which expansions the interactions are computed on the different tree levels. Here each box corresponds to a single multipole expansion, furthermore named “FMM box”.

In figure 2.3 it is assumed that exactly one layer of boxes around the examined box forms the near field. In the FMM the number of layers forming the fields can vary. The so-called “separatedness” defines where the near field ends and the

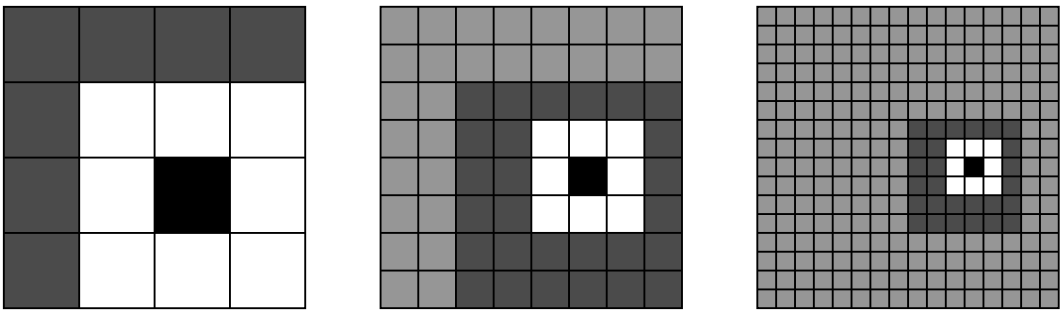


Figure 2.3: Let the examined particle be in the black box. Now the interaction sets for this box on different tree levels can be seen. On each level the interactions with the multipoles of the dark grey boxes are computed. The interactions with the light grey areas have already been computed, the white boxes must be excluded at this level due to convergence demands.[11, p. 394]

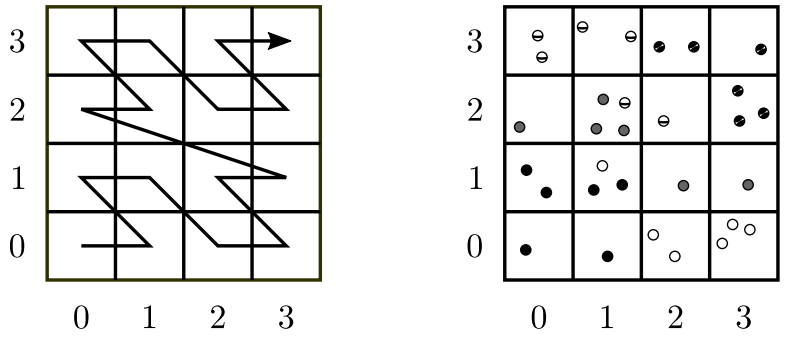
far field begins. Within the near field particle-particle interactions are computed, within the far field multipole-multipole ones instead. Therefore the separatedness has to be well balanced because a too big near field means a lot of particle-particle interactions with a compute complexity of $O(M^2)$ if M particles are within this near field. A too big far field has the effect of an enormous administration overhead because a lot of tree levels have to be created. Hence the separetedness has to be balanced for an optimal runtime of the algorithm.

To sum up the free parameters of the FMM, there are the tree depth, thus how many levels of multipoles are computed, the separatedness and the length of the multipole expansion.

The developed framework provides the possibility to display FMM specific properties like the FMM boxes. Other characteristics can be implicitely visualised. If for example the particle colour is chosen respective to the CPU which computed the particular particle, it is noticeable that for FMM computed data sets, particles close to each other have been computed by the same CPU. The reason for this are the space-filling curves.

A space-filling curve (SFC) is needed to assign the particles within the decomposed domains to the CPUs for a parallel simulation. A useful variant of these curves to be used is the Z-order curve (Morton-curve) because it is neighbourhood-preserving and the SFC indices can be easily calculated using bit shifts which is one of the fastest operations on a CPU. Figure 2.4(a) demonstrates the use of this space-filling curve to distribute the decomposed domain to five processors. For the Morton-curve the index of a box consists of the indices of this box in each dimension.

To give a two-dimensional example how the curve is created, let x_i and y_i be the binary digits of the box coordinates. The box with the box indices $(x_1x_0) = x_1 \cdot 2^1 + x_0 \cdot 2^0$ and $(y_1y_0) = y_1 \cdot 2^1 + y_0 \cdot 2^0$ gets the index $(y_1x_1y_0x_0) = y_1 \cdot 2^3 +$



(a) Z-Curve of the Boxes

(b) Distribution of the Particles to the CPUs

Figure 2.4: The first *Z-order curve* shows the Z-curve of the boxes. It is used to distribute the particles within the boxes to five CPUs which is displayed in figure 2.4(b). Each colour stands for another CPU.[11, p. 372, 375]

$x_1 \cdot 2^2 + y_0 \cdot 2^1 + x_0 \cdot 2^0$. The number of binary digits in the dimensions depends on the highest possible index which has to be represented.

To give a numerical example, the box with the coordinate indices $x = (2)_{10} = (10)_2$ and $y = (1)_{10} = (01)_2$ gets the index $(0110)_2 = (6)_{10}$, whereat the indices start with 0. The so created Z-curve defines the order of the boxes. This can be seen in figure 2.4(a).

Now the particles are ordered by the index of their boxes in a first step. Afterwards, all particles within each box are sorted among themselves using another Z-curve. This way the order of the particles is unambiguously defined so that they are distributed to the different CPUs. In the example of figure 2.4(b), 30 particles are assigned to five CPUs so that the first CPU gets the first six particles, the second one gets particle seven to twelve and so on.

The Z-order curve is only one possibility for a space-filling curve. Different space-filling curves like the Hilbert-curve can be used to assign the decomposed domains to the CPUs.[6, 7, 11][16, p. 85ff]

2.1.3 Physical and Simulation Properties

For the development of a visualisation framework for particle data sets, it is important to know the properties which are connected with particle simulations. All properties can be divided into two main groups, namely physical as well as simulation properties. Physical properties belong to the particles themselves. They also exist in real experiments and not only in simulations. Simulation properties originate from the calculation on supercomputers and do not only belong to particle simulations but to all kind of simulations. The following tables 2.1 and 2.2 explain selective representatives of both groups.

Physical Property	Dimensionality[1]	Explanation
Number of Particles	$\mathbb{N}$	This is the total number of all simulated particles. It can exist once for the entire data set if it is the same for all time steps. If it varies for each step, it exists anew for each time step.
Simulation Size	$\mathbb{R}^n$, $n \in \{1, 2, 3\}$	Also known as "periodic box size", the extension of the repeating box in periodic data sets. It only exists for periodic data. n is the periodicity. If $n = 0$ the data is non-periodic.
Charge	$\mathbb{R}$	
Mass	$\mathbb{R}$	
Velocity	$\mathbb{R}^n$, $n \in \mathbb{N}$	If the dimensionality is greater than one, the vector implies the direction of the property. The most frequent dimensionalities are one- and three-dimensional whereat the one-dimensional property often can be interpreted as the norm of a higher dimensional vector.
Density	$\mathbb{R}$	
Energy	$\mathbb{R}$	
Forces	$\mathbb{R}^n$, $n \in \mathbb{N}$	see the explanation of "Velocity"
Gradients	$\mathbb{R}^n$, $n \in \mathbb{N}$	see the explanation of "Velocity"
Fields	$\mathbb{R}^n$, $n \in \mathbb{N}$	see the explanation of "Velocity"
Pressure, Stress	$\mathbb{R}$	
Position	$\mathbb{R}^n$, $n \in \mathbb{N}$	Most times three-dimensional
Box shape		Shape of the simulated area

Table 2.1: This table contains some representatives of *physical properties* with the dimensionality of their data vectors and short explanations.

[1] of the data vector

Simulation Property	Dimensionality[1]	Explanation
Number of CPUs	$\mathbb{N}$	This is the number of processors used to execute the program on a supercomputer. Another possible value is the number of threads per node, so how many threads have been started on each compute node. The values are mostly powers of two.
Index	$\mathbb{N}$	To identify the particles within the different time steps, unique indices can be applied to them. For each particle, the index is equal in all time steps.
Runtime	$\mathbb{R}$	The time the simulation needs from program start to program end. It also can be given explicitely for the different sections of the calculation, for example an own measurement for the simulation of each time step.
Timestep	$\mathbb{R}$	The simulation works with data sets integrating the data over the time. The simulations do not calculate the chronological sequence of all physical properties continuously but rather for discrete time steps with time lags of a defined length between.
Efficiency	$\mathbb{R}$	In general, efficiency is a measure of the economy of a program concerning the use of ressources, time and storage space needed to solve the problem. In parallel programming, it means the ratio between the speedup[2] $S(p)$ and the number of processors p: $E(p) = \frac{S(p)}{p}$
Precision, Accuracy	$\mathbb{R}$	This is the grade of approximation used for input, calculation and output for each parameter. Dealing with numerical values, it represents the number of significant digits.

[2]The speedup is a measure of how much faster a problem can be solved if it is calculated in parallel with p processors compared to the fastest sequential program. The speedup $S(p)$ for p processors is $S(p) = T_S/T_P$ with T_S as the sequential and T_P the parallel execution time.

Simulation Property	Dimensionality[1]	Explanation
Memory Consumption	$\mathbb{N}$	The memory consumption is the amount of storage space needed to store the intermediate and final results of a program.
Domains	$\mathbb{R}^n$, $n \in \mathbb{N}$	For the simulation, a domain decomposition is needed assigning subdomains to the different CPUs so that each processor computes mostly its local domain.
Interaction Sets		To reach a complexity of $O(N)$ instead of $O(N^2)$, the domain is divided into near field and far field. The separatedness defines how large both fields are. It also controls the ratio of the computing time for both sets. For more information see 2.1.2.

Table 2.2: This table contains some representatives of *simulation properties* with the dimensionality of their data vectors and short explanations.

2.2 Visualisations

2.2.1 What is a Visualisation?

The first question to be asked when developing a visualisation framework is how a visualisation is defined. In general it is the illustration of measured data or the results of simulations. It interpretes this data on a graphically descriptive way displayed on screens or projection systems or plotted using printers.

The purpose of visualisations is the capture of large data sets to facilitate the identification of changes and trends. The adequate form of the visualisation depends on the type of data, its amount, the purpose of use and the competence and experience of the potential viewers. Possible forms are amongst others diagrams and multidimensional areas containing geometric figures. [3]

2.2.2 Why are Visualisations needed at all?

Dealing with the development of a visualisation framework, another question that may be asked is why visualisations are needed at all in the context of simulations.

The first answer to this question is that the results of simulations running on supercomputers are most times huge files with countless information stored therein. To check if the computation is correct and to analyse the results, mostly an exemplification of the output data of the simulation is needed. This way it is easier and faster to examine the computed data. Another aspect is that often the output files are written in binary mode which cannot be read directly by the user but only with the help of parsing programs. A visualisation can parse them easily.

Another answer is that for publications or talks pictures, diagrams and even films are more impressive and easier to understand than extracts from huge data files. A visualisation generates these illustrations. On poster sessions which are popular at scientific congresses, coloured and detailed pictures attract the attention of the visitors more likely than simple notes and columns of data. Moreover, complex structures can be explained better by pointing at them in illustrations.

To sum up, visualisations are indispensable in the scientific world to illustrate and exemplify huge data sets, complex structures and processes. They are easy to understand and may focus one's attention to a talk, publication or poster.

2.2.3 Existing Visualisation Tools and Frameworks

Before starting the development of a new framework, it is always advisable to evaluate if an already existing framework complies with all requirements. In the following, different existing visualisation tools and frameworks are briefly introduced stressing the way the input data can be processed and visualised in each case.

OpenGL

OpenGL is an interface independent of platform and programming language. It can be used for the development of three-dimensional computer graphics. The interface standard contains about 250 different commands to create complex three-dimensional scenes in real-time. The framework has been developed by Silicon Graphics and was first published as IrisGL in 1992. Today the OpenGL Architecture Review Board (OpenGL ARB) defines the OpenGL standard. The current version in June 2011 is version 4.1. Companies like Apple, AMD/ATI, Dell, IBM, Intel, 3DLabs, SGI and Sun participate in the OpenGL ARB.

OpenGL is a procedural graphics API[3]. The programming model is designed to define not the scene with the OpenGL commands but the steps necessary to achieve a certain appearance or effect. Each single step is in this case one of the OpenGL commands.

There are different groups of commands. First, there are some methods to draw three-dimensional graphics primitives like points, lines and polygons. Moreover, there are calls to define and manipulate lightning, shading, texture mapping, blending, transparency and other effects. There are no commands for tasks like window management, user interaction or file input and output. For these functionalities of a program, other libraries have to be used. GLUT[4] can be used for the window management, the standard libraries of the programming language often provide file access.

Concerning the particular implementations of the OpenGL standard, there exist two different approaches, namely generic software implementations and hardware implementations. In figure 2.5 these two variants are compared. There are some similarities between both: The application program uses services of the operating system as well as input and output services. A graphics device interface is needed to create output on the screen calling functions of the windowing system. In both cases, the OpenGL commands are called directly from the application.

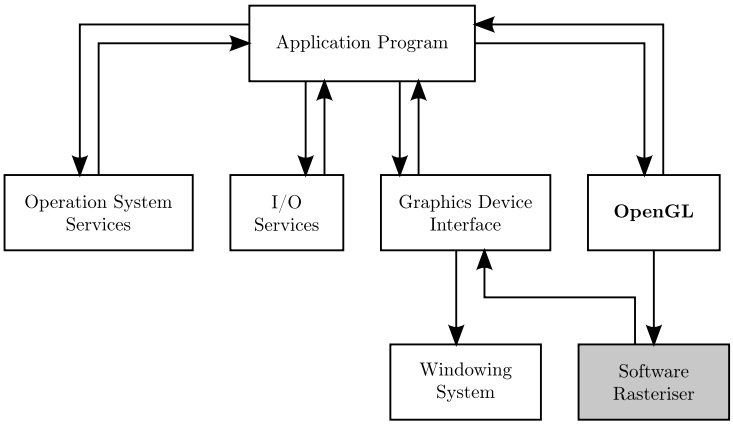
The main difference between both types of implementation can be formulated as follows: In software implementations, OpenGL uses a software rasteriser to construct the image. This rasteriser then calls commands of the graphics device interface but does not directly communicate with the windowing system. In hardware implementations, the OpenGL calls use services of the hardware driver which can communicate directly with the windowing system.

Software implementations are more portable than hardware implementations since they are independent of the built-in graphics card of the computer. Hardware implementations are faster because they may benefit from the special command set of the graphics card.

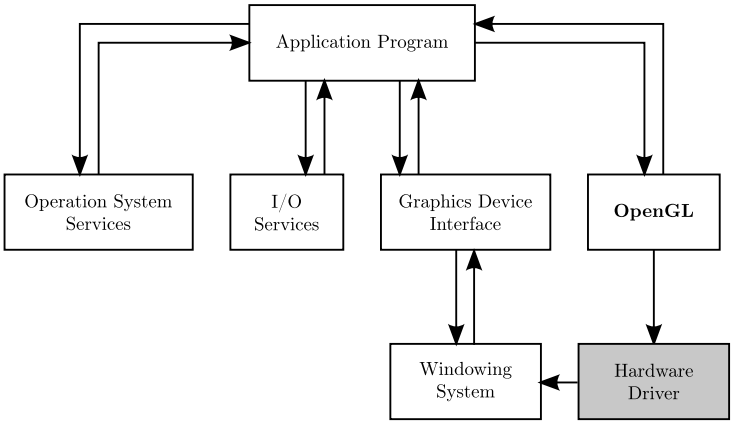
The processing of OpenGL commands is done in the following way (see figure 2.6). The called OpenGL commands are stored in a command buffer. It contains

[3] Short for “Application Programming Interface”

[4] OpenGL Utility Toolkit, a windowing system



(a) Software Implementation



(b) Hardware Implementation

Figure 2.5: The main difference between *software and hardware implementations* is the construction of the images by the software rasteriser on the one hand and the hardware driver on the other hand.[15]

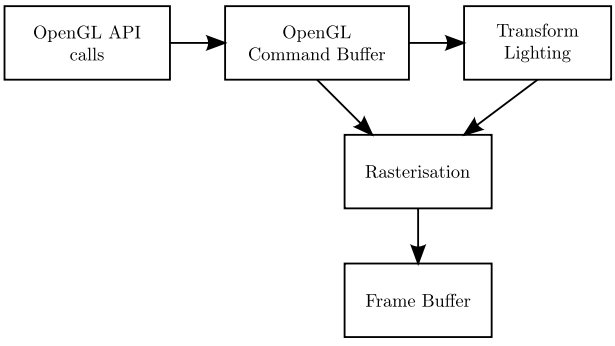


Figure 2.6: This pipeline scheme shows how OpenGL commands are processed.[13]

all information about the scene to be displayed. If the buffer is flushed, the data is passed to the next stage.

The transform and lighting unit recalculates the points used to describe the geometry of the objects so that the given location and orientation are correct. Furthermore, the lighting part calculates the correct colours for each vertex[5]. This step is computationally intensive.

The rasterisation unit creates the two-dimensional screen image from the geometries, the given colours and the textures. The image is placed in the frame buffer, which is the memory of the graphics display device, so that it will be displayed on screen.

To sum up, OpenGL is a framework providing low-level commands used to create two- or three-dimensional images out of graphical primitives. It is platform independent. Implementations for most popular graphic cards exist. Since it does not provide services for aspects like window management, always further APIs have to be used together with OpenGL.[13, 15]

ViSTA

ViSTA is a framework developed by the virtual reality group of Rechen- und Kommunikationszentrum of RWTH Aachen. It "is a software platform that allows integration of VR technology and interactive, 3-D visualization into technical and scientific applications"[4]. This means that with ViSTA in a first approach three-dimensional visualisations can be created which can then be used in virtual reality. Virtual reality (VR) systems provide a three-dimensional display as well as different ways of interaction with this artificial environment. Sophisticated systems even use "sensors on the user's body to sense movements that are then interpreted by the system as movements in the simulated world"[1]. So ViSTA provides methods to create interactive three-dimensional scenes.

The framework uses different other frameworks and toolkits to provide further features, for example OpenGL, the scenegraph library OpenSG or the windowing system GLUT. This way ViSTA is an API on a higher level than OpenGL. It has been tested on Windows and Linux platforms and is prepared for Macintosh at the moment. The framework provides a C++ interface.

The main advantage of ViSTA regarding OpenGL is the possibility to switch between different types of displaying systems by simply changing some parameters in configuration files. So the same application can run on desktop systems as well as on stereoscopic displaying systems (see also 2.2.4). Moreover, it is possible to call OpenGL commands within the ViSTA system so that the runtime advantages of OpenGL can be combined with the variability concerning the output media of ViSTA.

A disadvantage is that the system is still in development so that with new versions the interface sometimes changes. Therefore all existing applications have to

[5]The points defining the geometries are called "vertices" (singular: "vertex").

be adapted to the new one if the version has to be upgraded. In addition, the installation is not as easy as with OpenGL because all underlying frameworks have to be installed in advance.[4, 27]

ParaView

ParaView is an application designed for data analysis and visualisation. It is open-source and available for multiple platforms. It is developed by Kitware to analyse large data sets interactively in three-dimensional visualisations or using batch features. It uses distributed memory resources. ParaView is designed as an application framework so that the components can be reused for own tools.

The program can run on parallel and non-parallel systems under the operating systems Windows, Mac OS, Linux and Unix. The parallel platforms IBM Blue Gene and Cray Xt3 Unix are supported. For computations and rendering it uses VTK[6]. The user interface is written with Qt[7]. The aims of this framework are to provide an open-source and platform independent visualisation tool with an open, flexible and intuitive user interface. The architecture is extensible based upon an open standard.

ParaView provides, amongst others, the following features of visualisation:

- Remote of the visualisation: GUI[8] and visualisation can be run on different systems
- Data types: structured, unstructured, polygons, images, multi-block and others
- Processing operations (filters): create own data sets, the result of each operation is a new file
- Plotting vector fields using scalar or vectorial components for the vector orientation and length
- Creation of isosurfaces and contours using scalar or vectorial components
- Definition of cutouts and clippings using planes, threshold criteria or volumes of interest
- Generation of streamlines using integrators
- Displacing of points
- Calculation of new variables out of existing ones
- Python programmable filters, which can process the data
- Different possibilities to define subsets of the whole data to be focused
- Various data sources and filters

To sum up, ParaView is a program with a lot of practical filters which can be used for many different use cases. It is also extendible for further features.[17]

[6] Short for "The Visualization Toolkit", see http://www.vtk.org/
[7] http://qt.nokia.com/
[8] Graphical User Interface

VisIT

The visualisation program VisIT is a free parallel tool for the visualisation and analysis of scientific data which can run under Windows, Mac OS and Linux. It generates visualisations, animates them through time, manipulates them and saves the results for presentations. The program can display scalar or vector fields defined in $\mathbb{R}^2$ or $\mathbb{R}^3$ for structured or unstructured grids. According to the developers it can visualise data sets up to terascale size. VisIT provides amongst others the following features which can be used via a GUI.

- Plots: visualisation of data including boundary, contour, curve, label, mesh, pseudocolour, scatter, streamline, subset, surface, tensor, vector and volume
- Operators can be applied to the data to define subsets, e.g. slices, index selections and isosurfaces
- Time-based animations
- Qualitative and quantitative analysis and visualisation, for example the calculation of new fields out of existing ones
- Different grid types in 2D and 3D
- Parallel and distributed architecture: engine and viewer can run on different computers, the engine can be run in parallel
- New user interfaces can be added if they are written in C++ or Java
- Batch mode in Python
- Extensible via dynamically loaded plugins
- Tool to create plugins

To conclude, VisIT is like ParaView a visualisation and analysis tool providing a lot of features. It can be extended for not yet implemented use cases. It is not as well-known as ParaView.[19]

2.2.4 Demand for a new Framework

As described in the previous section, with ParaView and VisIT already two tools exist which provide a lot of features for data analysis and visualisation so that one may ask why another framework is needed.

To answer this question, in a first approach the title of this thesis has to be considered: A visualisation framework for "large particle sets" is needed. Both ParaView and VisIT can visualise this kind of data but they provide a lot of other features, too. This makes it difficult to become acquainted with these programs. For users which have never worked before with these visualisation tools, it is hard to find the needed features without taking courses in advance. So the first requirement is that the framework has to be specialised for large particle data sets so that the handling is easy.

The second reason to develop a new framework instead of extending an existing one is that the performance can be bad, when using visualisations of large data sets

on standard workstations. Both tools provide the possibility to be run on parallel architectures but also on workstations and notebooks sometimes a visualisation is needed. The existing tools are not specialised for particle data so that with a new tool the data storage can be optimised for this type of data in order to reach a better performance also on workstations.

Except for these two reasons, namely an easier handling of the program and a better performance by an optimisation for particle data sets, there are some other reasons why a new framework is needed. They all have to do with special requirements concerning the input and output of the program. These requirements are described in the following, all dealing with the idea to provide flexible interfaces for the "data exchange" with the environment of the program.

Requirements for the Input Data

The input of a visualisation tool consists of two different parts, the input data and the GUI interactions. There are different scenarios concerning the way how the input is communicated to the program. Most data sets have been computed on supercomputers. They can be stored on storage servers or on local computers in files so that the visualisation program can read these files. Alternatively the simulations stream the computed data live from the supercomputer so that the visualisation has to process the stream. This scenario is called online-visualisation. The visualisation framework shall provide both input types.

There is also a special requirement concerning the GUI. It shall be possible to run the visualisation on one computing system and the GUI on another one. For example in case of talks, it can be reasonable to let a graphic computer visualise the data onto a projection screen without a framing GUI so that the audience can focus on the displayed images. In this case the GUI may be run on a tablet computer, a notebook or even on a smartphone allowing the speaker to walk freely around and keep contact with the audience. In another case with the user sitting in front of the computer both the visualisation and the GUI shall be displayed on the same screen. Therefore the framework has to be modular to provide the possibility of a remote output display.

ParaView already provides the feature of a remote visualisation but only for the implemented GUI so that for example the use case of a GUI running on a smartphone is not realised by this framework.

Another important aspect is that the data format within the files or streams should be flexible so that the user simply has to define once how the data is aligned. For example in case of ParaView, an extension has to be written if the given format has not yet been implemented. This procedure is time-consuming if the user only wants to check if the simulation results are correct.

To sum up, there are three main requirements concerning the input : The loading of the input data has to be flexible to allow files and streams while also the data format can be easily specified. Furthermore, a remote output shall be available so that the GUI and the visualisation may run on separate systems.

Requirements for the Output Devices

For the output of the visualisation, an important requirement exists. There are a lot of different output devices available for visualisations. To list up only some of them, there are notebooks, desktop computers and stereoscopic projection systems (see figure 2.9). In ascending order these three types of systems normally have properties more adequate for visualisation issues than the previous one.

Furthermore, there are a lot of different devices used for the interaction with the displayed scene to create a virtual reality. Dealing with stereoscopic projection systems, there are three-dimensional input devices like gyrostick (see figure 2.7) and spacenavigator (see figure 2.8). In case of systems with (optical) tracking also tracked devices are possible. The visualisation tool shall not only provide the use of different output devices but also of the different interaction devices. The effort to switch between the different devices has to be minimised for the user.

ParaView supports neither three-dimensional interaction devices nor tracking systems.

Conclusion of Requirements

To sum up the demands for the visualisation tool, the already existing ones like ParaView and VisIT are usable but they do not provide the required flexibilities and requirements because of the requested specialisation on large particle data sets. Therefore a new framework has to be developed that is adapted to the requirements listed in the previous sections.

To provide the different features concerning the input, the framework has to be modular so that it may be extended as needed. The communication between the GUI and the output has to be abstract so that a remote output and different devices for the GUI are applicable.

The switch between different output and interaction devices can be achieved using a framework like ViSTA already providing this specific feature. In combination with OpenGL, a simple and fast display of the particle data can be realised. Therefore the output can be perfectly adapted to the requirements of particle data sets. Nevertheless, the framework must not be strapped to underlying frameworks so that all parts can be exchanged if required.

Figure 2.7: This gyrostick has got tracking targets used for optical tracking (see figure 2.9). It works as a 3D-mouse with integrated gyroscope to measure the velocity and angles of movements.[12]

Figure 2.8: This interaction device is a spacenavigator. It is a six-dimensional mouse supporting panning, rotation, curling and zooming.[2]

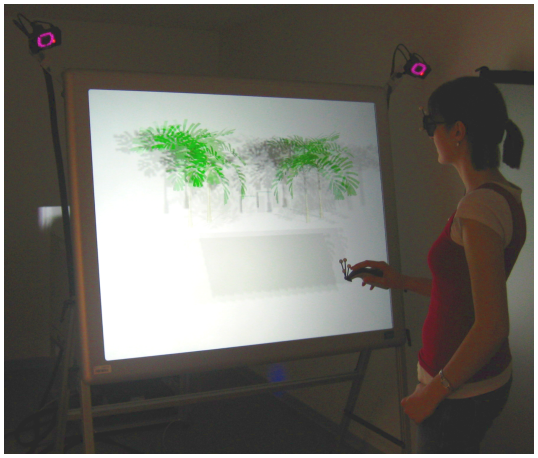


Figure 2.9: Stereoscopic projection system with optical tracking. The displayed visualisation is published as an example of a ViSTA application.

3 Realisation of the Framework

3.1 General Aspects

3.1.1 Choice of the Programming Language

At the beginning of a software development, some general aspects have to be clarified. The first one is the choice of the programming language used for the implementation thus limiting the number of libraries which can be used by the framework to be developed.

In principle, all object-oriented languages can be used since the framework has to be modular. Furthermore other toolkits and frameworks have to be considered. Most of them are provided for a few different languages. However, in some cases it is possible to create hybrid programs with different programming languages because for example Java can call C functions which can in turn call Fortran routines. Also C functions can be called out of C++ classes.

ViSTA is provided for C++ only. For OpenGL, there exist implementations in different languages, amongst others C and Java. Other toolkits that may be included like VTK[1] are also provided in Java and C++. This leads to the decision between C++ and Java. Often Java is slower than C or C++ since the program is executed in the context of the additional virtual machine. Because ViSTA is a good possibility to provide access to different output devices and other frameworks like OpenGL and VTK, C++ will be used for the implementation of the new visualisation framework.

3.1.2 Project Documentation

The project has to be well documented so that decisions concerning the implementation are comprehensible and new programmers can become acquainted with it easily. Mainly the functionality of all classes, their methods and attributes have to be described during all phases of the development and implementation.

To facilitate the embedding of the inline source code documentation into the project, a reasonable way is to use *doxygen*[2]. Therefore all classes, structures, methods and attributes have to be documented in a special way: The comments start with `/**` and end with `*/`. Using special tags like `@author`, `@param` and `@return`, the comments can be structured. With `@param` the input parameters of functions are documented, `@return` marks the return parameters.

[1] http://www.vtk.org/

[2] http://www.stack.nl/ dimitri/doxygen/

The tool *doxygen* parses the source files, detects these special comments and translates them into different output formats like HTML[3] or LaTeX. In a configuration file amongst other aspects it can be defined which output types shall be generated. The created LaTeX code can then be included into a LaTeX document describing the entire project. Each user can generate the documentation by himself with the simple call to "`doxygen`".

3.1.3 Build Environment

For the distribution of the visualisation framework it is important to provide a build system. Again it has to be evaluated which build environments are already given by the possibly used toolkits. For ViSTA it can be chosen between *make*[4] and *cmake*[5]. OpenGL can be built in the same way as each other C or C++ program. For the GUI, a toolkit like Qt[6] can be used for which also a *cmake* structure exists.

Hence it is reasonable to provide a *cmake* structure for the framework to be developed. This tool has the great advantage that the structure of *cmake* files has to be created once. New files have to be added to this structure later. To install the project, a single call to *cmake* is sufficient.

[3] Short for "HyperText Markup Language"
[4] http://linux.die.net/man/1/make
[5] http://www.cmake.org/
[6] http://qt.nokia.com/

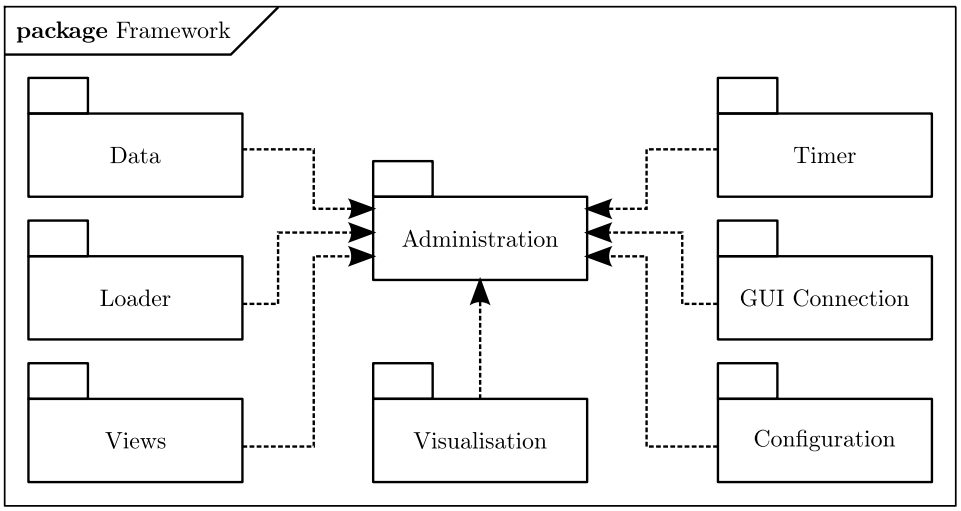


Figure 3.1: The UML Package Diagram shows the modular structure of the visualisation framework.

3.2 The Data Model

3.2.1 Modular Concept

Structure

As worked out in section 2.2.4, all parts of the visualisation framework have to be exchangeable. To give some examples, different ways of input have to be available like file and stream input. Furthermore, different output and interaction devices have to be supported. Dealing with different output devices varying from notebooks to high-end graphical systems with stereoscopic projections, also different GUIs are needed since in some use cases a remote output is reasonable.

In principle, all parts of the visualisation system have to be exchangeable. For each module an interface is needed defining the provided methods and services.

Figure 3.1 displays the different modules of the entire visualisation framework. In the following sections all modules are briefly described highlighting their tasks and functionality. A more detailed dependency graph of the whole framework can be found in appendix A.

Module “Administration”

The “Administration” module forms the centre of the framework. It initialises all other modules and communicates with them during runtime. In the initialisation step it connects a module to all modules it has to communicate with. For example the “Visualisation” and the “Data” modules have to be connected because the data is needed to visualise. To add new modules to the framework or to exchange an existing one, the modifications have to be implemented here.

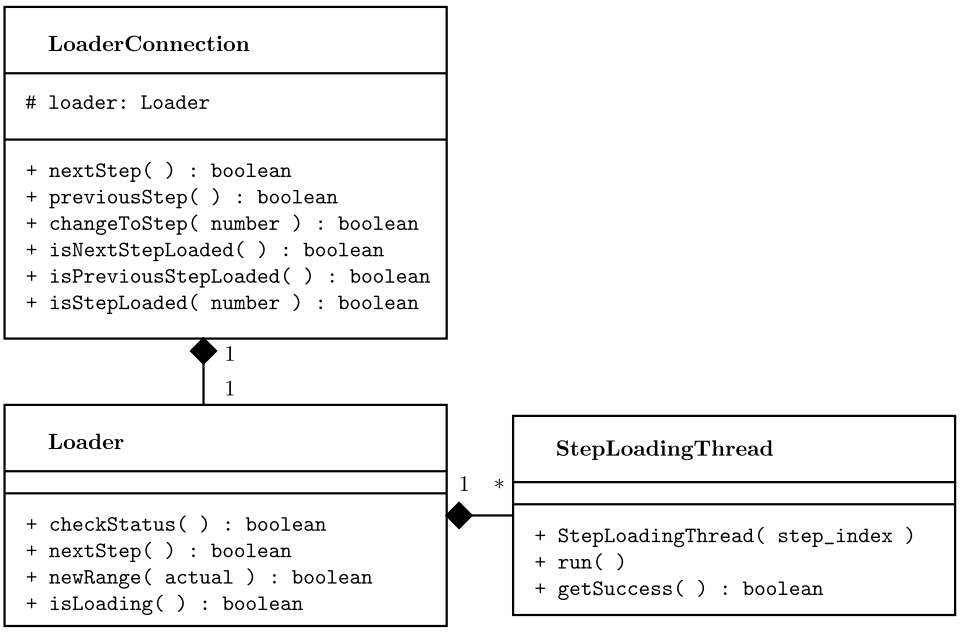


Figure 3.2: The UML Class Diagram summarises the functionality of the module "Loader". The access point for other modules is the class `LoaderConnection`.

Module "Data"

The main task of the "Data" module is the storage and administration of the read input data. The data sets consist of several time steps holding the information of all particles existing in the simulated region.

The full data set can contain up to thousands of time steps with up to billions of particles. Visualising the data, a locality of time is assumable. This means that after the visualisation of time step t_k, most probably either step t_{k-1} or step t_{k+1} will be visualised. Therefore it is reasonable not to store all time steps of the data set but always a set of time steps around the current one. Therefore the "Data" module has to manage the stored steps.

This module is independent of all other modules. It provides methods to store new particles and time steps. Since the framework is multi-threaded by design, it is important to synchronise the access to the stored data. Detailed information about the data storage is available in section 3.6, page 65.

Module "Loader"

The module "Loader" has to preload the next time steps so that always a set of time steps is stored neighbouring the currently visualised one. The loading process for each new time step runs in a separate thread so that the visualisation does not have to be interrupted.

The UML class diagram in figure 3.2 displays the internal structure of this module. The class `LoaderConnection` acts as the access point for other modules. The classes `Loader` and `StepLoadingThread` are hidden by this interface. This access class provides the methods `nextStep()`, `previousStep()` and `changeToStep()`. They first check if the requested step is already stored in the data structures. In this case the requested step is marked as the one to be visualised. If the step is not yet available, it is loaded in advance. After marking the step, the set of neighbouring steps is loaded adaptively.

The interface class delegates the loading to its `Loader` object. If the methods `nextStep()` or `newRange()` are called, this object creates a new instance of `StepLoadingThread` with the index of the step to be loaded. This newly created thread runs in parallel to the rest of the framework and reads a single time step. The `getSuccess()` method can be used to check if it was possible to load the step since for example the corresponding input file might not exist.

The "Loader" module needs access to the "Data" module to access the loaded steps and to the "Configuration" module to identify where the data has to be loaded from and in which format it is stored.

Module "Views"

The "Views" module administrates the different views to be visualised. A more detailed description of the view concept can be found in section 3.3, page 34.

To explain it shortly, the views of the visualisation framework are comparable to views of databases. In this case a view is a specialised extract of the entire database, hence a subset of the entire data. Dealing with databases, they are provided so that each user is allowed to work only with the data he is permitted to.

Figuratively, the views of the visualisation framework consist of the choice of a subset of the entire data with a defined way to display it. For example all particles with a negative charge are chosen and visualised as points in a three-dimensional scene. Alternatively, all particles within a defined area are selected whereat their total energy is plotted in a two-dimensional diagram.

This module does not visualise the data but describes how it shall be visualised. This way the underlying toolkits used for the visualisation are encapsulated and can be exchanged easily without side-effects.

The "Views" module has to communicate with the "Data" module for the selection of subsets and with the "Visualisation" module to display the data.

Module "Visualisation"

The module "Visualisation" encapsulates the underlying visualisation toolkits and frameworks from the remaining modules. Its main task is to carry out the visualisation of the views which are defined in the "Views" module. This module directly

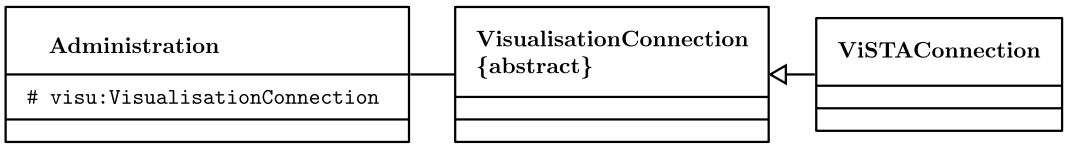


Figure 3.3: The visualisation framework ViSTA is encapsulated using the interface `VisualisationConnection` providing all methods needed to visualise the data. The "Administration" module holds a reference to the abstract but not a special type.

calls methods of visualisation toolkits like for example ViSTA or OpenGL. How this encapsulation is realised, is demonstrated in figure 3.3.

To enable exchangeability of the underlying visualisation frameworks, the "Visualisation" module provides the abstract class `VisualisationConnection` as an interface to all possible visualising components. The class `Administration` as part of the "Administration" module has got a reference to an object of this abstract but not to a special type. The class `ViSTAConnection` represents the implementation of the visualising part using ViSTA and OpenGL to display the data. To use for example VTK instead, a class `VTKConnection` can be implemented which also derives from the interface. In the initialisation phase of the system, now an object of this new type has to be created instead of the type `ViSTAConnection`.

The abstract class defines which methods have to be provided by the visualising part of the framework. These are for example methods to add new views or to change global parameters of the visualisation like the background colour. So inheritance as a central part of object-oriented design[20] is used to reach exchangeable modules.

Module "Timer"

The particle data sets contain several time steps. The visualisation of time-dependent data can be compared to a movie which consists of about 20 frames displayed per second so that the human eye gets the impression of motion. The same needs to be done in this project. Here it is possible to change regularly from a time step to the following one whereat the delay between two steps is user-defined.

The "Timer" module has the task to observe the delays of a fixed size between two time step changes. It has to inform the "Loader" module to change to another time step to be visualised if the time lag is expired. Also the "GUI Connection" module has to be informed of the change so that the displays within this interface can be updated.

For the time management, another thread is needed which can be stopped and restarted in case the user stops the animation to examine a time step in detail. The defined time lag is not a fixed value but has to be understood as a minimum delay

between two changes, since dealing with large time steps the preloading of the next steps takes a longer time than possibly defined short delays.

Module "GUI Connection"

The "GUI Connection" module defines the message protocol for the communication between the framework and an adapted GUI implementation. It does not provide an implementation of the GUI but the methods to inform it about changes or to request user inputs.

The access point for the communication with the user interface is implemented to expect a TCP/IP[7] connection. This way a remote output is possible but also both the GUI and the visualisation can run on the same system.

Module "Configuration"

The module "Configuration" provides classes to read and store all configuration and input files except for the data input. To give an example, there are files defining global visualisation parameters like the background colour, the format of the data input files. Moreover, there are files defining the views. Examples for all files can be found in the appendices B and C of this thesis.

All input files except for the data files are written in XML format. Therefore a class `XMLReader` is provided by this module containing static methods to validate and parse all of these files. More information about the XML files can be found in section 3.2.2.

Another class of this module is the `Logfile` used to write messages, warnings and errors into a defined file so that the program can be started outside of a console. Otherwise the output could not be seen by the user.

3.2.2 Storage Model

Auxiliary and Configuration Files

To use the framework, a lot of parameters have to be configured in advance. To keep it user-friendly, all configurations have to be storable for future uses so that it is done only once. Therefore it is reasonable to let the user write down the configurations in files of a defined format so that these files can be reused.

To use the configuration files in the framework, they have to be validated and parsed. Therefore a file format has to be used for which this can be done without great effort. A practical file format is XML which stands for "Extended Markup Language". Such files contain a tree structure with a single root. All nodes of the tree are marked with tags enclosed with < >.

[7] Short for "Transmission Control Procotol/Internet Protocol", a communication protocols used in the internet

To validate an XML file, an XSD file has to be written once. XSD stands for "XML Schema Definition". These files are also written in XML defining all valid specifications. Each XML file references the XSD file it implements. For a lot of programming languages, libraries exist to validate an XML file against its schema. Therefore the validation of the configuration files can be done with little effort.

To parse the configuration files, the so-called DOM can be used. This is an abbreviation for "Document Object Model" which provides methods to process the XML file tree node by node. Implementations for a lot of programming languages including C++ exist. This model also provides ways to modify existing files or to create new ones so that the framework can save modifications of the configuration for future uses. In addition, the files can be created by a GUI thus simplifying the process of configuration.

Examples for the different XML files used by the framework can be found in appendix B.

Different Input Sources

There are several ways for the origin of the visualisation data. The simulations can either stream the output directly for an online visualisation. Alternatively, the results are stored on storage servers. In the second case, there can be a separate file for each computed time step or a single file containing the particle data of all time steps.

In both cases, the parsing of the input has to be done in a separate thread for three main reasons: First, loading another step must not result in an interrupt of the visualisation. Both the visualisation and the loading of steps can run in parallel if different threads are used. While a new step is loaded, the old one is still visualised.

Second, loading the steps in a separate thread provides the opportunity to preload steps before they are needed. This way the delay between two changes is minimised.

The third reason is that in online visualisations it has to be waited for the next step to be simulated before it can be displayed. If the visualisation has got a separate thread, the user can interact with the scene while the "Loader" module waits for the next portion of data.

Different Output Targets

The preferred output of the visualisation is on a standard monitor screen. It can be the screen of a notebook or desktop computer or even a (stereoscopic) projection screen. This type of output is also adequate for talks and presentations. For other purposes, like publications or websites, pictures or movies are needed but not interactive programs. Therefore it is reasonable to provide a possibility of screen capturing for movies and screenshots to provide pictures of the interactive program.

Screenshots are provided by a lot of visualisation frameworks, also by ViSTA, so that this feature is automatically included in the newly developed framework. For

screen capturing, a lot of tools exist which can be used to capture the whole screen or parts of it like for example VLC[8] or Wink[9].

3.2.3 Interfaces for Inter-Module Communication

Dealing with a modular framework, it is important to look at the inter-module communication. All modules provide interface access points for the communication with the other ones. Figure 3.4 displays the most important interfaces of the modules presented in section 3.2.1 and the main communication directions. The names of the interfaces are the names of the implemented classes.

The `Administration` interface has to communicate with all others to initialise the framework. It acquaints the interfaces among themselves.

The `TimeManagement` object has to inform the `LoaderConnection` object of changes between the time steps so that the "Loader" module can load the required time step if necessary. Then the notified object uses the internal objects of its module to load a new time step. Therefore it needs the information stored in the `InputConfiguration` object. It adds the loaded step to the `Timesteps` object. Furthermore it notifies the `VisualisationConnection` object to update the data.

To visualise the data, the `VisualisationConnection` object respectively the inner objects of the "Visualisation" module have to fetch the particle data from the `Timesteps` object and general configurations for the visualisation like the background colour from the `CommonConfiguration` object.

The views managed by the `ViewAdministration` object fetch the particle data from the `Timesteps` object. The operators of the views evaluate this data to create the subsets.

The user can add new views, modify an existing one or remove one from the visualisation using a GUI. It sends corresponding messages to the `GUIConnection` object. This notifies the `ViewAdministration` object of the changes which in turn notifies the `VisualisationConnection` object.

3.2.4 External Interfaces

In the previous section the internal interfaces of the framework have been described. This section deals with the questions which interfaces are provided by the framework to be used from the outside and which interfaces of other frameworks have to be included. Dealing with the used external interfaces it is important how they are encapsulated so that the developed framework is not completely dependent of any of them. To give an example, if the support of a framework has been ended, still existing errors will not be fixed so that it might be necessary to use another one instead.

[8] http://www.videolan.org/vlc/

[9] http://www.winkstreaming.com/en/wink_player/

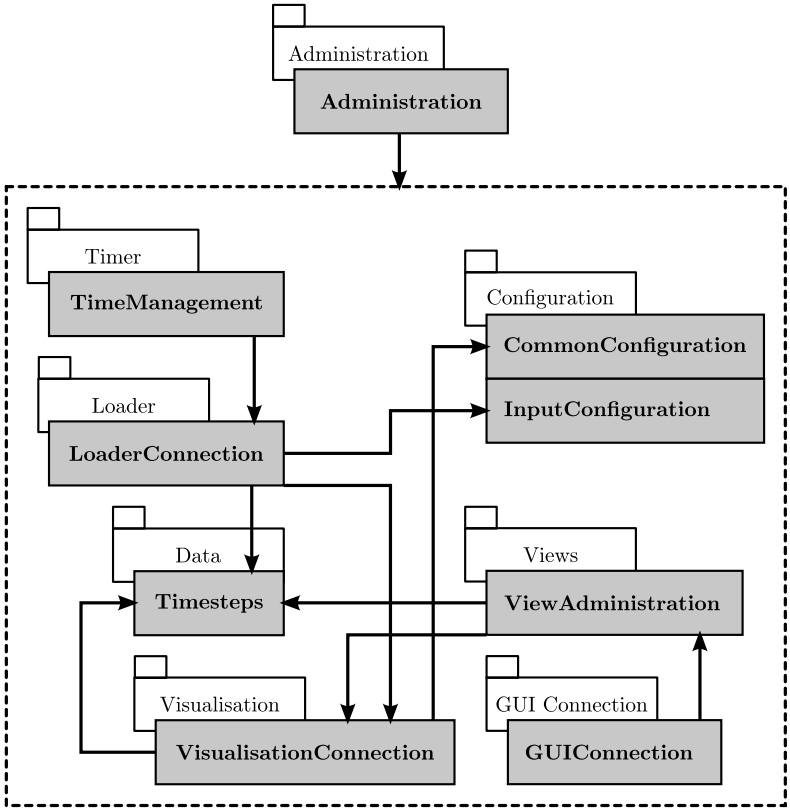


Figure 3.4: The most important interfaces of the different modules are displayed including the main communications. Note that the "Configuration" module provides two interfaces.

Provided Interface

The developed visualisation framework provides one single interface to be used from outside, namely the `GUIConnection` object as an access point for a GUI implementation. This leads to the question why a user interface has not yet been implemented.

To answer this question, the different use cases of the visualisation framework have to be examined. There are two main cases: On the one hand, the user works with a desktop system where both the visualisation and the user interface are displayed on the same screen. On the other hand, a remote output device can be used so that the visualisation runs on a projection screen while the GUI is executed on a portable device, e.g. a tablet PC or a smartphone.

The GUI implementations for these two approaches have to fulfil demands due to the different operating systems and programming languages available on the used GUI output device. Therefore no user interface implementation is provided but an access point and a message protocol. The `GUIConnection` object waits for a GUI to communicate using TCP/IP.

Utilised External Interfaces

In the first implementation, there are three different interfaces used by the developed framework, namely Qt, ViSTA and OpenGL.

Interface	Usage
Qt	First, the Qt thread framework is used to create different threads and to synchronise parameters using the provided mutex[10]. Moreover, the DOM and XML implementations are used.
OpenGL	OpenGL is used for the visualisation of the different main and overlay views.
ViSTA	From ViSTA, the window management is used as well as the support of different output and interaction devices. ViSTA provides interfaces to draw directly using OpenGL. This way the combination of both frameworks is possible without additional effort. The good performance of OpenGL is combined with the device and window management of ViSTA.

Table 3.1: These external interfaces are used by the developed framework.

[10] A mutex is a mechanism to synchronise the access to so-called critical sections which must not be reached by more than one thread simultaneously.

3.3 Concept of Multiple Views

3.3.1 Why is a Concept of Multiple Views Needed?

The developed framework is a "Multiple View" framework. This leads to the questions what is meant by the term "View" and why a concept of views is needed at all.

To answer these questions, a closer look at the data to be visualised must be taken. The particle data sets simulated on supercomputers contain up to billions of particles in each time step. Even if all particles are displayed as single pixels, nevertheless, the user will only see a pixel cloud. This way it is impossible to analyse the data optically. Hence, it must be possible to choose subsets of the entire data set to be visualised afterwards. Moreover, the user is often interested in questions like the following ones:

- How many particles cross a particular plane?
- How does a cluster of particles behave after an external force acted on it?
- How is a special subset, e.g. all atoms with negative charge, distributed and how does the distribution change over time?

To sum up, it must be possible to define data subsets. On one hand, a visualisation of all particles cannot be analysed due to the size of the input data. On the other hand, in some cases the user is interested in the visualisation of a particular part of the simulated data. Therefore a way is needed to define these subsets.

If the subsets have been chosen, several possibilities to display them are imaginable. First, there are three-dimensional scenes with the particles visualised as points or spheres or points with attached attributes like vector arrows. Alternatively, the tree structure of the FMM (see section 2.1.2, page 8) surrounding the particles can be displayed instead. In addition, two-dimensional plots of the data sets are possible in form of cutting planes displayed in a two-dimensional scene or in form of diagrams. For example the diagrams may display the course of the total energy of the system. Thus, after defining a data subset, it has to be defined how this subset is visualised.

These two tasks, the choice of a subset and the definition of the way to display it, are called a "view". The choice of data is done by an "operator" (see section 3.4, page 42), the definition of its display is formulated in a "filter" (see section 3.5, page 58). Figure 3.5 illustrates this correlation. The terms "operator" and "filter" are chosen according to ParaView and VisIT (see section 2.2.3, page 14). VisIT calls the mechanism to define a subset of data "operator, in ParaView the processing of data, so its display, is called "filter".

In most cases, the display of several views at the same time is needed to analyse the data properly. To give an example, a use case can be to display all particles crossing a defined plane in a three-dimensional view. At the same time the total energy of the system is plotted against the time in a diagram. Furthermore, simulation

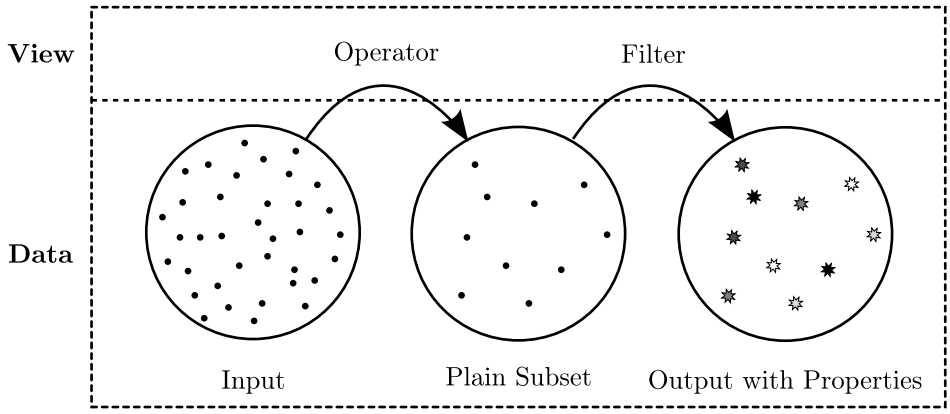


Figure 3.5: A view consists of two components, an operator and a filter. The operator chooses a subset of the whole data set, the filter describes how this subset is visualised.

properties like the number of the displayed timestep and the number of used CPUs are displayed textually. Therefore, the framework provides the possibility to define a couple of views to be visualised at the same time.

Both parts of the view, the operator and the filter, are defined independent of each other. This way they can be reused easier. In addition, it is possible to choose a subset of data with an operator and to apply a couple of different filters to it. This way the particles can be visualised as points in a three-dimensional scene defined by the first view. The filter of the second view determines to create a diagram out of the same data plotting the total energy of the subset. In this example, the operator has to be defined only once. This procedure is also possible the other way round, so the same filter is adapted to different operators.

As already indicated, it has to be differentiated between two groups of views concerning the types of the included filters. On the one hand, there are filters describing to display the particle data in a three-dimensional scene. The views owning to these filters are called “main views”. On the other hand, different filters describe two-dimensional displays, namely two-dimensional cutting planes, diagrams or textual output. The views containing these filters are called “overlay views” because they are displayed in form of overlays covering parts of the three-dimensional scene. Both types of views are described in the following.

3.3.2 Main Views

Main views contain a filter describing the visualisation of particle data in a three-dimensional scene. If a set of main views is chosen to be visualised, they are all included in the same three-dimensional scene. This is demonstrated in figure 3.6.

The particles can be visualised in different ways. The fastest way with respect to the performance is to display a pixel for each particle. Here the pixel colour can

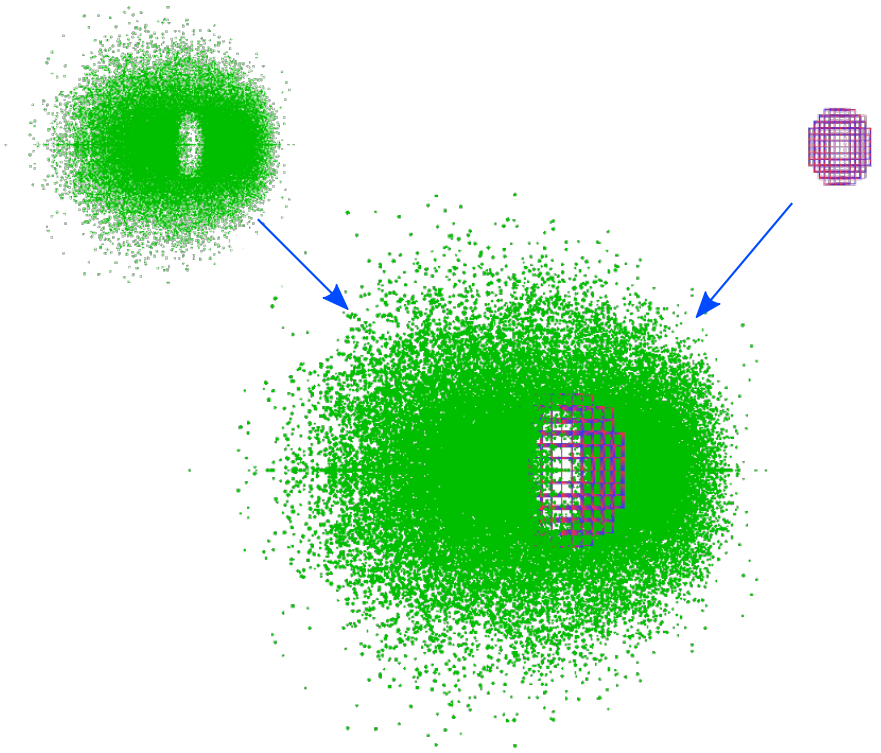


Figure 3.6: Here two main views are combined in the same 3D-scene. One of the used operators chooses all particles within a sphere, the other all outside of it. The views use different filters. The cut out sphere contains a dense particle cluster.

be adapted to a physical property of the particle so that this property is visualised together with the particle position. A second physical property besides the colour can be applied to the particle size.

Alternatively to the pixels, the particle data can be visualised as spheres while again colour and the sphere radius are adapted to physical properties. It is also possible to display a vector field with arrows attached to the points. In this case properties with up to three components can be used to define the orientation of the arrows.

An abstract way to visualise particles in a three-dimensional scene is to display the boxes of the FMM (see section 2.1.2). In this case not the particles themselves but all boxes containing at least one of the particles are displayed. Such a display can be important if the distribution of particles is of interest but not the position of each single particle.

Further ways to visualise particle data in a three-dimensional scene are possible. More detailed information about the different types of filter can be found in section 3.5, page 58.

The administration of main views does not require additional actings besides the insertion and deletion of objects representing the respective views to the `ViewAdministration`. To define a view, the user has to include an XML file defining the operator and filter. The precise structure of these files is illustrated in appendix B.

3.3.3 Overlay Views

Difference to Main Views

Each main view contains a filter defining to visualise the data in a three-dimensional scene. They are all included in the same scene, thus displayed in the same window. Overlay views contain different types of filters. In contrast to the main views, they are added at the borders of the window overlaying the main views. So the difference between main and overlay views is the way to display the view either included in a three-dimensional scene or as a cover of this scene. Figure 3.7 illustrates this window composition.

Different Subtypes

There are different subtypes of overlay views with respect to the type of the included filters. A first possibility is the textual display of status data. This can be physical properties like the number of particles or simulation properties as for example the index of the time step or the number of CPUs used to compute this step.

Another possibility are two-dimensional scenes. These can either be cutting planes or the full scene reduced to two-dimensionality so that one of the particle position coordinates is ignored for the display.

A different class of two-dimensional displays are plots. The plot displays simulation or physical properties over time. An example could be to visualise the development of the total energy over time.

Also three-dimensional scenes may be visualised as overlays, e.g. to display a subset of the data visualised in the main scene. This mode of display may be confusing, mainly if similar filter parameters are set for the overlay and the main views so that the user cannot identify to which of the scenes for example a point or sphere belongs to. Hence, three-dimensional scenes as overlays have not yet been added to the system due to practical reasons. However they can be added in future versions.

Adjustment of Overlays

The administration of all views depends a lot on the type of the view. In case of main views, they simply have to be added to the program and have to be introduced to the visualisation module. The administration of the overlays is more complicated because the user must have the possibility to define which overlay is placed where. In the following it is described how this has been realised.

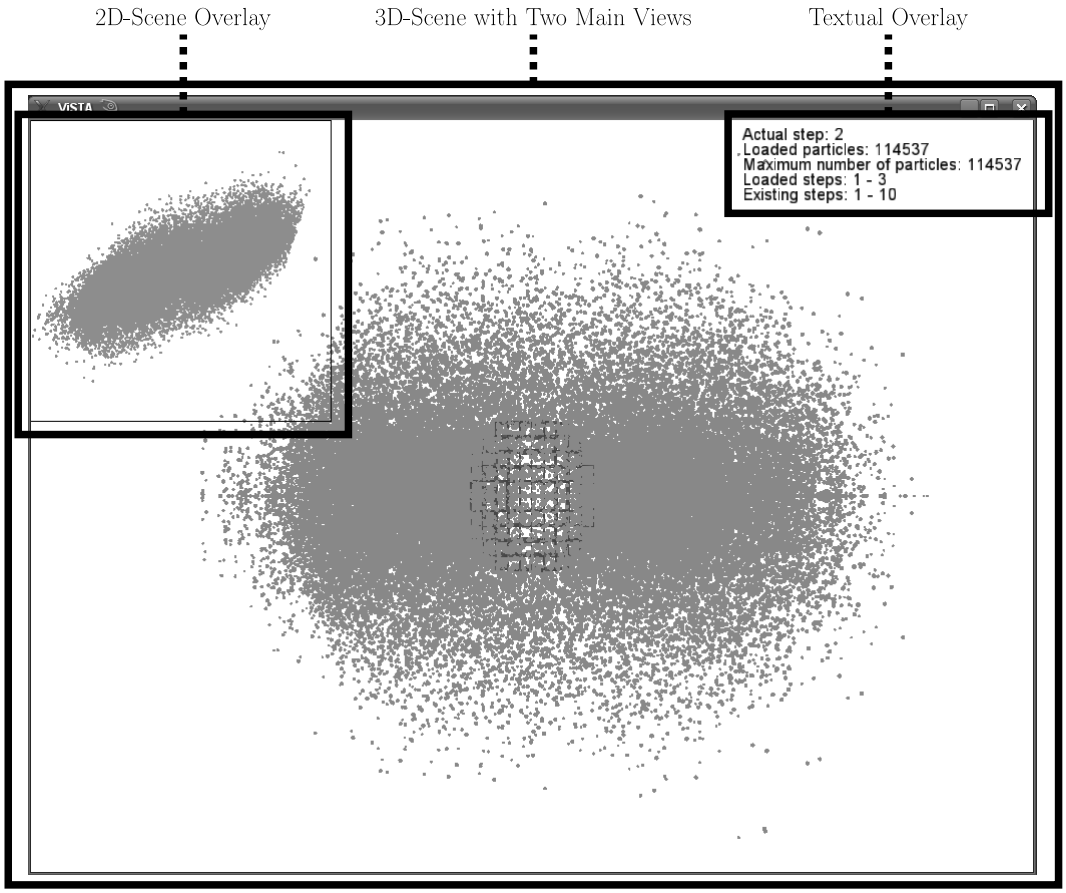


Figure 3.7: This sample window contains two different main views integrated in the 3D-scene and two different overlay views covering this scene. One of the overlays is an informal text, the other one displays the particle data in form of a 2D-scene looking alongside the negative y-axis.

First, all overlays form a frame alongside the window borders. Their backgrounds have an adjustable transparency property so that the three-dimensional scene might show through. For each overlay it can be defined at which of the four borders it is positioned.

Moreover, the order of all overlays placed at the same window border has to be defined. Therefor each overlay gets an "order" parameter. This is a positive integer value. The overlay with the greatest order value is positioned first, ordering them from the left to the right respectively from top to bottom. Overlays can have the same order value. In this case they are arranged according to the order in which they were added to the framework.

Each overlay has got a width and height defined in pixels. Using these values and the window resolution, it can be determined if there is enough space for another overlay to be added. If one of the borders is overfilled with views, the views to be positioned last are clockwise moved to the next border. If all borders are completely

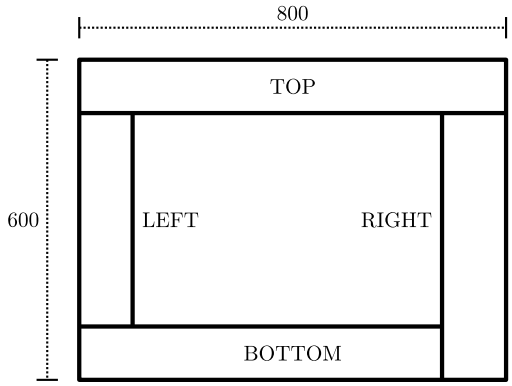


Figure 3.8: The illustrated window has a resolution of 800×600 pixels. The labels of the window borders are drawn in. Width respectively height of the borders are not yet fixed since these properties are determined by the size of the overlays.

filled and a new view has to be added, it will not be displayed.

All overlays are added next to their neighbours seamlessly. So the total width of all top or respectively bottom overlays is the sum of their widths, the total height of all sidewise overlays is the sum of their heights. Since the overlays do not need to have the same dimensions, they are all aligned to the window borders. The heights of the bottom and top borders are the maximum heights of the overlays aligned at the respective border. The widths of the sidewise borders are the maximums of the overlay widths. This information is needed to determine how much empty space is at each border.

To conclude, the adjustment of the overlays is defined by their size, the chosen window border, their order value and the values of these parameters of all overlays added before.

Example for the Adjustment of Overlays

The following example demonstrates the relation between the overlay size, the order and border parameters. In figure 3.8, the four window borders are named. The window has got a resolution of 800 pixels in width and 600 pixels in height. The figure also shows which corner belongs to which border.

If an overlay has to be added to the visualisation, the parameters width, height, order and the window border have to be defined in an XML view file (for the complete files see appendix B). A first overlay is to be added to the top border. In this case the order does not have any influence because this border is empty so that it is added right at the top left corner (see figure 3.9).

The second overlay also has to be added at the top border. So first of all, it has to be determined if there is enough free space. With a window width of 800 pixels and an overlay width of 200 pixels, there remain 600 pixels free which is enough

for this overlay. The order of the new overlay is greater than the one of the already added overlay so that the new one is added on the left of the old one (see figure 3.10). The height of the top border stays the same because the first overlay is higher than the second one.

Also the third overlay has to be added to the top border. Its height is greater than the height of the first and second added overlays so that the complete border is larger now (see figure 3.11). After adding this one, there are only $800-200-240-300 = 60$ pixels free with respect to the width of this border.

If now a fourth overlay wider than 60 pixels has to be added to the top border, one of them will be automatically moved to the right border. The fourth overlay to be displayed in this example is 160 pixels wide but shall be added to the top border. The result can be seen in figure 3.12.

This example demonstrates that the user can adjust the overlays on his own defining the order value and the window border. Alternatively, the adjustment can be done by the `ViewAdministration` automatically. Therefore the user gives all overlays the same order value and the same window border. The only parameters which have to be exact are the width and height of the overlay view.

```
1 <overlayView
2   border="top"
3   order="4"
4   width="200"
5   height="100" />
```

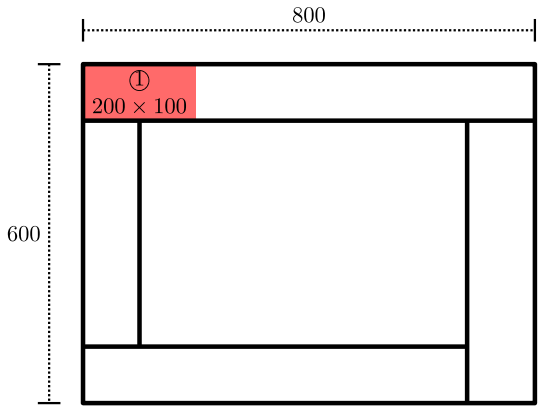


Figure 3.9: An overlay is added to an empty window border.

```
<overlayView
  border="top"
  order="6"
  width="240"
  height="80" />
```

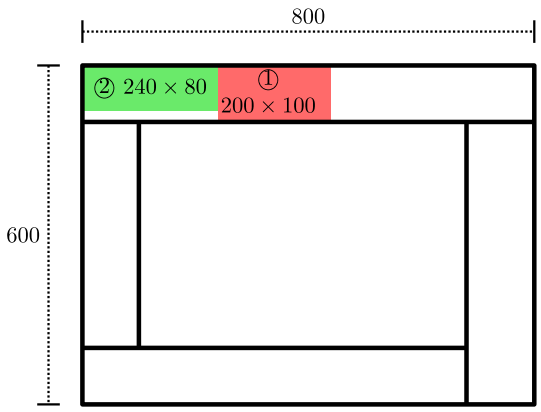


Figure 3.10: An overlay is added to the top border already containing another one with a smaller order. The circled numbers defines the order in which the overlays have been added.

```
<overlayView
  border="top"
  order="1"
  width="300"
  height="160" />
```

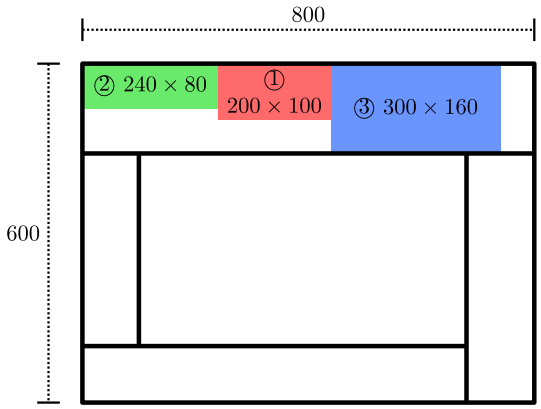


Figure 3.11: Another overlay is added to the top border. There already exist two with higher orders.

```
<overlayView
  border="top"
  order="1"
  width="160"
  height="160" />
```

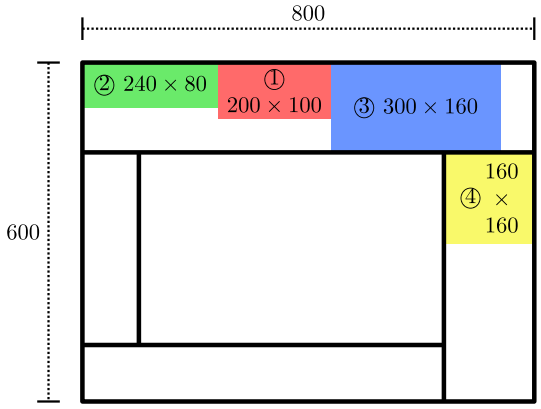


Figure 3.12: A fourth overlay has to be added to the top border for which only a width of 60 pixels is left. The new overlay is wider than these 60 pixels. It has the minimal order value and is moved to the right border.

3.4 Input Selection Via Operators

3.4.1 Why Do We Need Data Subsets At All?

As already suggested in section 3.3, the task of an operator is to choose a subset of particles out of a complete time step. This leads to the question why data subsets have to be visualised instead of the full data set.

To answer this question, there are three main reasons. The first reason is that a lot of data sets simulated on supercomputers contain up to billions of particles for each time step. If all these particles are visualised as simple small points, the display contains a huge point cloud. This makes it impossible to identify trends and changes within the data set from one time step to the next.

Moreover, on customary desktop computers, a visualisation of such a large data set has got a poor performance. Therefore it is needed to sort out particles so that the point cloud is less dense but still visualised qualitatively identical and the framerate increases.

Another reason is that in many use cases it is not needed to visualise all particles but to highlight certain properties. Examples for this case are the following ones.

- Display all particles which are in the near field of a defined box.
- Visualise all particles computed by a defined set of CPUs to check the load balancing during the simulation.
- Chose only particles crossing a given plane at time step t_i.
- Display all particles with a negative charge.
- Track a small set of particles whose development is of interest.

To conclude these examples, particles are evaluated concerning the emphasized property so that all particles fulfilling the defined criteria are chosen to be visualised, all other ones are pruned.

The last reason to create subsets is that sometimes single particles have to be tracked. This means that these particles are optically highlighted so that the changes of the particle position and its visualised physical properties can be examined. In this use case the rest of the particles can be completely hidden or visualised on a simple way.

To sum up, there are three main reasons why data subsets are visualised instead of the whole data, namely

1. to reduce large data sets to influence the performance and clearness,
2. to highlight certain properties or
3. to track single particles.

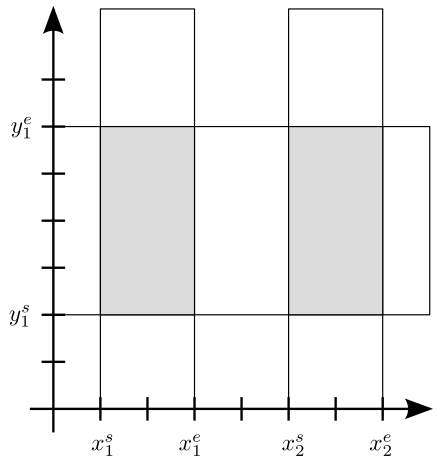


Figure 3.13: This diagram demonstrates the choice of particles by specifying domains of definition for each coordinate of the particle positions. Only particles within the grey highlighted area are chosen.

3.4.2 Selection Criteria

In the previous section it has been worked out that there are three main reasons why the definition of data subsets might be required. This leads to the question which selection criteria exist to define the subsets. In general, the selection of several particles out of the data set can be done evaluating physical and simulation properties like the particle position, its index or physical properties as for example the charge, mass or velocity.

Some simulation properties like runtime, efficiency or precision cannot be used for the choice since they are on an abstract level and refer to the simulation of a full time step. In the following, three different selection criteria will be described more detailed.

Selection Operator – Particle Position

Concerning the position, there are two general ways to choose particles. The first possibility is to define domains of definition for each axis. All particles whose coordinates of the position are within these domains will be chosen. In fact, the domain of definition for a single axis does not need to be a connected and continuous range but can consist of several ranges.

In the two-dimensional example in figure 3.13 the operator chooses all particles with $x \in [x_1^s, x_1^e] \cup [x_2^s, x_2^e]$ and $y \in [y_1^s, y_1^e]$. Here only the particles within the intersections of the domains of the x-axis and the ones of the y-axis are chosen.

The second possibility to choose particles using their position is to define simple geometric shapes. All particles within these shapes are then chosen by the operator. Examples for these shapes can be cuboids or spheres.

Cuboids are unambiguously defined by domains of definition for each of the

axes. A particle is within this shape, if for all three dimensions the respective coordinate of the position is within the domain of definition.

Spheres are uniquely defined by the coordinates of their centre and the radius. A particle with the position $P = (x_p, y_p, z_p)$ is within the sphere with radius r and centre $C = (x_c, y_c, z_c)$ if one of the equivalent inequalities (3.1) and (3.2) is fulfilled. The evaluation of the second formula might be faster since the power function can be easier computed than the root function.

$$|\overline{PC}| = ||\vec{C} - \vec{P}||_2 = \sqrt{(x_c - x_p)^2 + (y_c - y_p)^2 + (z_c - z_p)^2} \leq r \tag{3.1}$$

$$|\overline{PC}|^2 = ||\vec{C} - \vec{P}||_2^2 = (x_c - x_p)^2 + (y_c - y_p)^2 + (z_c - z_p)^2 \leq r^2 \tag{3.2}$$

Selection Operator – Particle Index

Another way to define subsets is to create a list of unique particle indices. All particles whose indices are within the list, are chosen by this operator. This list contains several domains of definition like $l = \{2, \{4 - 10\}, 15, \{22 - 24\}\}$. In this example all particles with the indices 2 or 15, indices between 4 and 10 or respectively between 22 and 24 are chosen.

The main use case of the index selection is particle tracking since this is the easiest way to address a particular particle. The particle position might change with each time step, physical properties cannot be used because they do not identify a particle unambiguously.

Further Properties as Selection Criteria

The third possibility to define subsets is the most complex one, using the characteristic of a certain physical or simulation particle property. This type of choice works only with simulation properties which belong to each single particle like the CPU which computed the particle.

The property has to be named via its unique ID. Then domains of definition can be defined analogue to the variant using the particle position.

To give an example for this type of choice, all particles could be chosen whose charge is negative or which are computed by the CPUs 2, $4 - 6$ or 10. Concerning the vectorial velocity, all particles with the z-component of the velocity smaller than 2.0 but greater than 1.5 could be used.

Usage of Reference Time Steps

After determining the particular selection criterion, another decision has to be taken because there are two ways to evaluate the operator. Either the criterion is evaluated anew for each time step or it is once evaluated for a reference time step and translated into unique IDs for adjacent time steps. In the second case, the indices of all particles chosen in this step are stored and visualised in all time steps.

In the first variant, all selected particles fulfil the selection criterion in the displayed time step. In this case some particles vanish from one step to the next while others appear. In the second mode, the movement of particles fulfilling the criterion once can be observed, for example particles with a defined start velocity are displayed during the entire time response although they might be slower or faster in other steps but the reference one.

Combination of the Criteria is Needed

In a lot of use cases, the needed subsets cannot be defined using the position or the index or a single property but a combination is needed, which is demonstrated in the following examples.

- All particles with a specific charge, e.g. marking them as electrons, *and* within a defined cuboid have to be chosen.
- Only electrons *or* protons need to be visualised.
- The particles computed by CPU 2 *or* 4 *and* a mass less then 0.5 are chosen.
- Only the particles within a huge cuboid *and not* within an inner sphere are to be displayed.

These examples show that the three different criteria have to be connectable among each other but also operators implementing the same criterion have to be combinable. So a way has to be found to define simple sub-operators implementing one criterion and to combine them in a second step. Common in all examples is that the different sub-operators are connected with phrases like “and”, “or” and “not”, so mathematical operators of propositional calculus. Therefore it is necessary to analyse how these operators can be used to develop a complex operator structure for the definition of data subsets.

3.4.3 Excursus: Propositional Calculus

Propositional calculus is a branch of logic dealing with the composition of propositions via (logical) connectives. In classical propositional calculus, each proposition has got one of the logical values true or false. The value of a complex expression can be determined evaluating the connectives using the particular values of their operands, the propositions. So the connectives are the operands of this branch of logic.

There are six different connectives, namely the negation $\neg$, the conjunction $\wedge$, the disjunction $\vee$, the implication $\Rightarrow$, the biconditional $\Leftrightarrow$ and brackets $()$ to define the order. The negation equates the colloquial “not”, the conjunction the “and” and the disjunction the “or” which have been used in the examples in section 3.4.2.

The following truth tables show the meaning of all connectives. A and B are propositions, 1 stands for true, 0 for false.

A	$\neg A$
1	0
0	1

A	B	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
1	1	1	1	1	1
1	0	0	1	0	0
0	1	0	1	1	0
0	0	0	0	1	1

Table 3.2: Unary and Binary Logical Operands

If an expression is true for each combination of propositions, this expression is called tautology. If it is false for all combinations, it is named contradiction. Iff two expressions E_1 and E_2 are equivalent, $E_1 \Leftrightarrow E_2$ is a tautology.

The examples in section 3.4.2 solely use negations, conjunctions and disjunctions, which leads to the hypothesis that these three connectives may already be sufficient to replace all others. In fact, the implication and biconditional are equivalent to an expression containing only these three:

$$(A \Leftrightarrow B) \Leftrightarrow ((A \wedge B) \vee (\neg A \wedge \neg B)) \tag{3.3}$$

$$(A \Rightarrow B) \Leftrightarrow \neg(A \wedge \neg B) \tag{3.4}$$

To prove that the implications in formulas (3.3) and (3.4) are tautologies, truth tables can be used (see tables 3.3, 3.4). The rows highlighted with a dark grey background are the rows to be checked for being tautologies.

A	B	$\neg A$	$\neg B$	$(A \Leftrightarrow B)$	$\Leftrightarrow$	$((A \wedge B)$	$\vee$	$(\neg A \wedge \neg B))$
1	1	0	0	1	1	1	1	0
1	0	0	1	0	1	0	0	0
0	1	1	0	0	1	0	0	0
0	0	1	1	1	1	0	1	1

Table 3.3: Truth Table for Equation (3.3)

A	B	$\neg A$	$\neg B$	$(A \Rightarrow B)$	$\Leftrightarrow$	$(\neg$	$(A \wedge \neg B))$
1	1	0	0	1	1	1	0
1	0	0	1	0	1	0	1
0	1	1	0	1	1	1	0
0	0	1	1	1	1	1	0

Table 3.4: Truth Table for Equation (3.4)

Since the implication and biconditional connectives can be replaced by expressions of negations, conjunctions and disjunctions, it is reasonable that also other

connectives like the *exclusive or* can be replaced the same way. For this connective the symbol $\dot{\vee}$ is used. Formula (3.5) contains the replacement of this connective.

$$(A \dot{\vee} B) \Leftrightarrow (\neg(A \Leftrightarrow B)) \Leftrightarrow ((A \wedge \neg B) \vee (\neg A \wedge B)) \tag{3.5}$$

So, the *exclusive or* can be replaced by a negated implication in a first step or directly using a disjunction of conjunctions, so a disjunctive normal form. Truth table 3.5 proofs this formula.

A	B	$\neg A$	$\neg B$	$(A \dot{\vee} B)$	$\Leftrightarrow$	$(A \Leftrightarrow B)$	$\Leftrightarrow$	$((A \wedge \neg B)$	$\vee$	$(\neg A \wedge B))$
1	1	0	0	0	1	1	1	0	0	0
1	0	0	1	1	1	0	1	1	1	0
0	1	1	0	1	1	0	1	0	1	1
0	0	1	1	0	1	1	1	0	0	0

Table 3.5: Truth Table for Equation (3.5)

In fact, all connectives can be replaced by an expression of negations, conjunctions and disjunctions. Furthermore, the combination of negations and conjunctions or of negations and disjunctions is sufficient to replace the third one so that the existence of two is enough[22, p. 71]. The reason to use all three connectives is that in most cases it is more intuitive to use an "or" respectively "and" instead of a more complex expression. Formulas (3.6) and (3.7) show the possible replacements which are proved with the truth tables 3.6 and 3.7.

$$(A \wedge B) \Leftrightarrow (\neg(\neg A \vee \neg B)) \tag{3.6}$$

$$(A \vee B) \Leftrightarrow (\neg(\neg A \wedge \neg B)) \tag{3.7}$$

These formulas are equivalent to de Morgan's law:

$$\neg(A \wedge B) \Leftrightarrow (\neg A \vee \neg B) \tag{3.8}$$

$$\neg(A \vee B) \Leftrightarrow (\neg A \wedge \neg B) \tag{3.9}$$

A	B	$\neg A$	$\neg B$	$(A \wedge B)$	$\Leftrightarrow$	$(\neg$	$(\neg A \vee \neg B))$
1	1	0	0	1	1	1	0
1	0	0	1	0	1	0	1
0	1	1	0	0	1	0	1
0	0	1	1	0	1	0	1

Table 3.6: Truth Table for Equation (3.6)

To conclude, using negations and conjunctions or disjunctions, all expressions of propositional calculus can be generated. Since it is intuitive to use directly

A	B	$\neg A$	$\neg B$	$(A \vee B)$	$\Leftrightarrow$	$(\neg$	$(\neg A \wedge \neg B))$
1	1	0	0	1	1	1	0
1	0	0	1	1	1	1	0
0	1	1	0	1	1	1	0
0	0	1	1	0	1	0	1

Table 3.7: Truth Table for Equation (3.7)

conjunctions (disjunctions) instead of the negated disjunction (conjunction) of the negated propositions, all three connectives have to be available to define operators as part of views.

Dealing with operators choosing data by property characteristics or indices, the intuitive way of speaking uses the words “and”, “or”, and “not”, so the propositional connectives. Using the particle positions, it is also possible to use the terms “union” and “intersection” instead of “or” and “and” because the inner imagination is to intersect or union certain areas.

Union and intersection are terms of set theory which studies sets instead of propositions. The connectives of propositional calculus have equivalents in set theory. Let A and B be propositions, M and N sets and U the basic set containing M and N. The following table 3.8 contains some equivalences between the two theories.

Propositional Calculus		**Set Theory**	
Term	Expression	Expression	Term
Negation	$\neg A$	$U \setminus M$	Complement
Conjunction	$A \wedge B$	$M \cap N$	Intersection
Disjunction	$A \vee B$	$M \cup N$	Union
Exclusive Or	$A \dot{\vee} B$	$M \triangle N$	Symmetric Difference

Table 3.8: Equivalences between Propositional Calculus and Set Theory

Dealing with the development of a framework, there is no difference concerning the evaluation of expressions of set theory or propositional logic. The difference is simply that in some cases the terms of one of the theories is more intuitive than the other one whereat the terms “and”, “or” and “not” are more familiar to most users than “conjunction”, “disjunction” and “negation”. Thus the framework has to offer the operands and, or, not, union and intersection to define the required complex data operators[22, p. 60ff (logic), p. 86ff (set theory)].

3.4.4 Implementation of Complex Expressions

After the data operators have been defined by the user, they have to be implemented in a programming language leading to the question how mathematical operators can be implemented generally since propositional connectives and set operations are nothing more but mathematical operators.

Each mathematical operator has got one or more operands. In complex expressions the operands can be operators in turn. In the expression $1 + (5 - 4)$ the $-$ has the operands 5 and 4, the $+$ has the 1 as the first operand, the second is the expression $(5 - 4)$.

This leads to the conclusion that mathematical expressions can be inserted into a tree structure[23, p. 331]. The leaves are (arithmetic) operands, all inner nodes are operators. Figure 3.14 shows the trees for the expressions $1 + (-3) \cdot (5 - 4)$ and $(A \vee B) \wedge (\neg C)$.

It has to be noted that for some operators the order of the operands does not matter. These operators are commutative like the $+$. For non-commutative operators like $-$ the order of the operands is important. For the tree this leads to the fact that the order of the child nodes is fixed for all non-commutative operator nodes.

To evaluate a tree, it has to be traversed post-order. This means that for a specific node first all child nodes are evaluated and at last the node itself. The child nodes are evaluated from the left to the right[23, p. 337]. So for non-commutative operators the first operand has to be in the left child node, the second one in the right node.

To sum up, mathematical operators can be inserted into tree structures with the inner nodes containing connective nodes and the leaves being choice nodes. To evaluate such a tree, it has to be traversed post-order interpreting the child nodes from the left to the right.

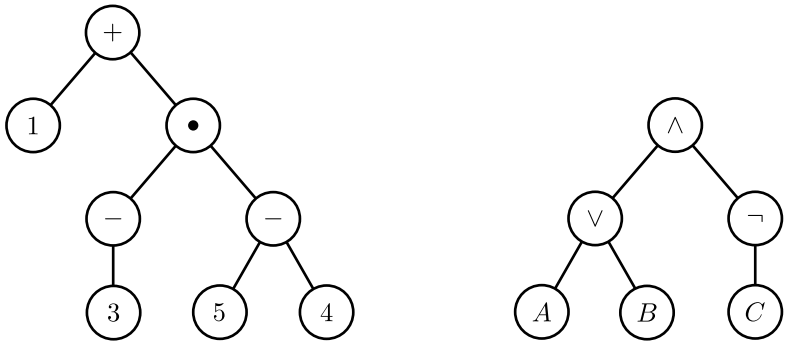


Figure 3.14: These trees contain the mathematical expressions $1 + (-3) \cdot (5 - 4)$ and $(A \vee B) \wedge (\neg C)$.

3.4.5 Definition of Operators in Input Files

Operators as part of views are defined in XML files so that they can be reused easily. In this case, the hierarchical structure of this file format fits perfectly the tree structure of mathematical expressions.

Within an operator, there are two different types of nodes. On the one hand, there are the connectives of propositional calculus, so basically “and”, “or” and “not”. These nodes are always inner nodes because they expect at least one child node. On the other hand, there are nodes which choose the data. They are called “choice nodes”. They are leaves of the tree because they cannot have child nodes.

Four different types of choice nodes are possible: They refer to the position, the index, scalar or vectorial properties. Scalars and vectors are distinguished separately since the parameters to be set in the operator file differ because of the dimensionality. In appendix B, the different choice nodes are described more detailed.

To create more complex operators, the choice nodes have to be combined using connectives. Typically, a negation represents an unary operator, conjunction and disjunction are binary, but in case of an operator tree, an inner node might have an arbitrary number of child nodes.

Given that the user defines the operators using the tree structure of XML files, there is no reason why conjunction and disjunction nodes must have only two child nodes. Moreover, the trees can be much more flat if the conjunction of three propositions can be done with a single “and” instead of a first “and” connecting two propositions and a second one connecting this result with the third proposition. Hence the “and” and “or” nodes must have at least two child nodes, the “not” node needs to have exactly one child.

Since each connective node can have another connective node as its child, this results in a (theoretically) unlimited tree. Therefore the number of operator links is also theoretically unlimited. In fact, a possibly restricting parameter is for example the maximum filesize.

The following example in figure 3.15 shows a more complex operator tree using connective nodes and choice nodes combined.

3.4.6 Developed Class Structure

In the previous section it has been analysed that the operators can be defined using the tree structure of XML files. These files have to be written by the user. The next step is to validate and parse these files. The validation can be done easily since an XSD file defining all valid operator XML files is provided by the framework.

Parsing an input file always consists of two steps, namely to read the file and to store the information in an internal object structure. So a class structure is needed which can fully describe all operators. The structure has to be extensible to keep it open for new choice and connective nodes. The framework provides the class structure displayed in figure 3.16, page 52.

```
<operator>
    <or>
        <and>
            <position>
                <sphere radius="0.5" >
                    <centre>0.5 0.5 0.5</centre>
                </sphere>
            </position>
            <scalar ID="MassID" >
                <domain>0.25 0.75</domain>
            </scalar>
            <not>
                <index>
                    <domain>50 100</domain>
                </index>
            </not>
        </and>
        <not>
            <and>
                <position>
                    <cuboid>
                        <x>0.0 0.4</x>
                        <y>0.0 0.4</x>
                        <z>0.0 0.5</x>
                    </cuboid>
                </position>
                <vector ID="VelocityID" >
                    <norm method="euklid" >
                        <domain>10 20</domain>
                    </norm>
                </vector>
            </and>
        </not>
    </or>
</operator>
```

Listing 3.1: Example of a complex Operator XML File.

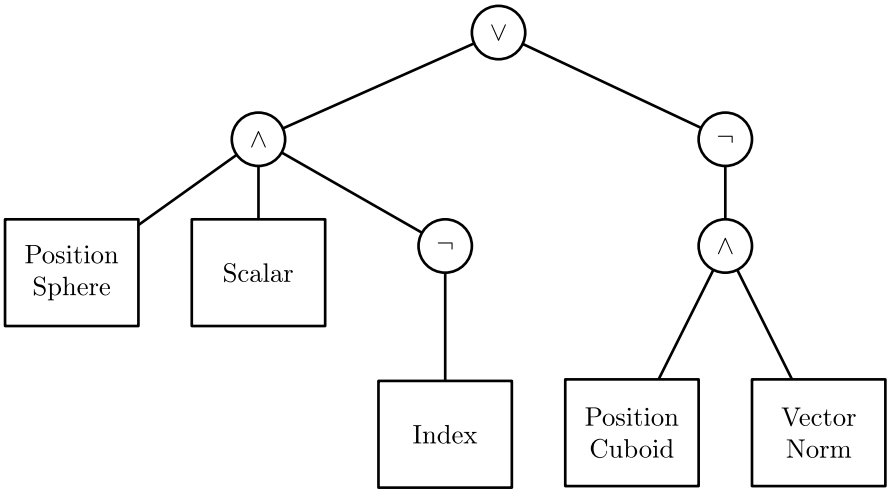


Figure 3.15: Tree Representation of the same complex Operator.

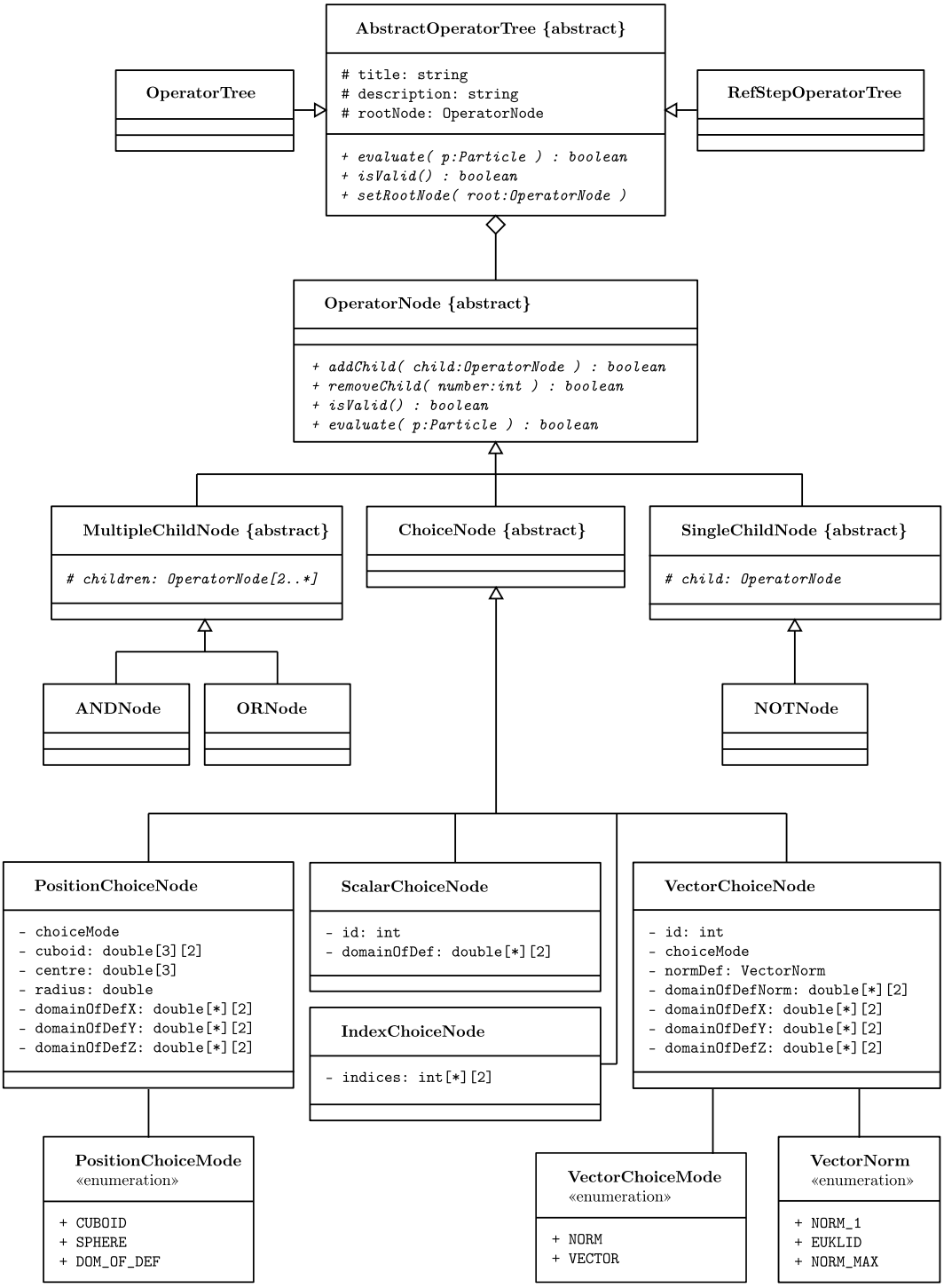


Figure 3.16: This UML class diagram contains the class structure used to store operators.

Tree Classes

The class `AbstractOperatorTree` is the interface for all operators. There are three main methods provided which have to be implemented. The method `setRootNode()` sets the root of the tree. The function `evaluate()` traverses the whole tree post-order to check if the given particle is chosen by the operator. The method `isValid()` checks if the tree can be evaluated. This is not possible if no root node has been set or if any node within the tree is incomplete. For example an "and" node is incomplete if it has less than two child nodes.

`OperatorTree` and `RefStepOperatorTree` are two implementations of this interface. The first variant evaluates the operator anew in each time step. The other one evaluates a reference time step first and stores the indices of the particles chosen in this step. For all other steps, a particle is chosen if it has been chosen in the reference time step. If the method `evaluate()` is called, both implementations call the `evaluate()` function of the root node and return this result.

Node Interfaces

The class `OperatorNode` is the interface for all nodes in the operator trees. No node is directly derived from this interface but from one of three abstract node types. `MultipleChildNode` is the abstract type for all connective nodes which can have two or more child nodes. `SingleChildNode` is the equivalent for connective nodes with one single child node. `ChoiceNode` is the abstract implementation for all nodes choosing particles because of their characteristics.

Both `MultipleChildNode` and `SingleChildNode` already implement all methods except for `evaluate()` because the behaviour is the same for all derived types. In case of the multiple variant, the `addChild()` method has to be called twice or more to make the node valid since it cannot be evaluated with less than two child nodes. So `isValid()` returns true if two or more child nodes exist. In case of the single variant, `addChild()` adds a child node only in case that none existed before. `isValid()` returns `true` if the node has got exactly one child node.

Connective Node Types

`ANDNode` and `ORNode` are implementations of a `MultipleChildNode`. Their `evaluate()` methods call the methods of all child nodes combining the results with the respective connectives "and" or "or". The `NOTNode` is an implementation of `SingleChildNode`. Its `evaluate()` method calls the method of the child node and negates the result.

Choice Node Types

The classes `PositionChoiceNode`, `ScalarChoiceNode`, `IndexChoiceNode` and `VectorChoiceNode` are implementations of `ChoiceNode`. They

provide the four different selection criteria to choose the subset which have been described in section 3.4.2, page 43. They store the information needed to apply the criterion to the particles. For example to choose the particles because of their indices, a list with the domains of definition of all valid indices has to be stored in the attribute `IndexChoiceNode.indices`. The abstract class `ChoiceNode` already implements the methods `addChild()` and `removeChild()` since the behaviour is the same for all implementations.

Extension with New Nodes

If the class structure of operators shall be extended with a new node type, first it has to be examined from which node interface this one has to be derived, e.g. the connective *exclusive or* has to be derived from `MultipleChildNode`.

Now the respective class has to be added implementing all methods not defined by the interface. Furthermore, the XSD file describing all valid operators has to be extended so that operators using this new node type can be validated. At last, the class parsing the XML files has to be extended. All other parts of the framework are not affected by the extension.

To conclude, the XML operator files can be parsed into the given class structure which is extensible so that new selection criteria or connectives can be added easily. The effort consists of extending the XSD file used to validate the operators, writing the new node class directly derived from `ChoiceNode`, `MultipleChoiceNode` or `SingleChoiceNode` and implementing the import of the new node type in the reader class used to parse the XML files. Thus, theoretically unlimited trees are definable.

3.4.7 Optimising the Expressions

Operators together with filters are a central concept of the framework. Each operator has to be evaluated for every particle when the visualisation changes between two time steps. Hence, non-optimally defined operators can result in a bottleneck of the framework. Therefore one has to check carefully if there is unused optimisation potential. The “optimal” operator must be equivalent to the given one but requires a smaller computing time for the evaluation.

Given that the user sets up the operator files, they will be defined intuitively, which does not have to be the optimal definition for the evaluation. So the framework may check all operators for a possible optimisation after parsing them from the input files. There are two possible approaches for optimisation: restructuring the child nodes of connective nodes and the use of tautologies of propositional calculus.

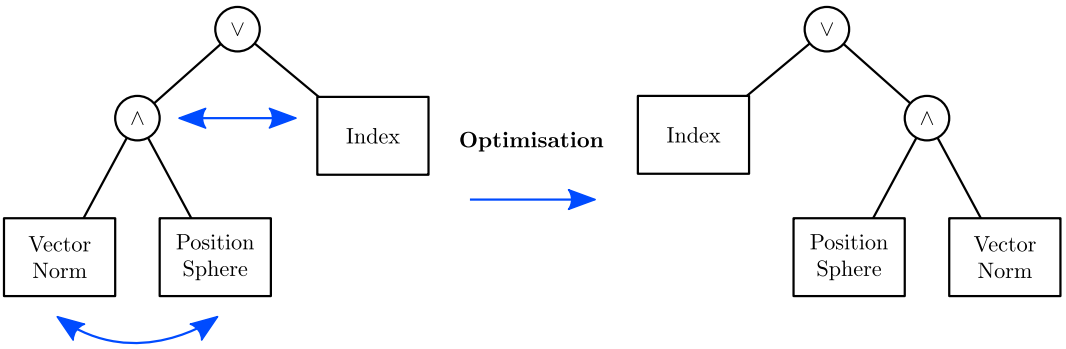


Figure 3.17: This optimisation is based upon the different complexities of the nodes.

Different Complexities of the Nodes

The first approach bases upon the fact that the evaluations of the different choice node types have different complexities and that all connectives are commutative. To give an example, for the evaluation of a position node with the mode "sphere", the effort is always the same. The evaluation of an index node however strongly depends on the list of domains of definition because the index has to be compared with all list elements.

Most programming languages stop the evaluation of logical expressions as soon as the result is unique. They evaluate the operands in reading direction. So an `or` construct returns true as soon as one of the operands is `true`, an `and` construct is false as soon as the first `false` operand has been detected.

Thus it is possible to reorder the child nodes so that the ones with the smallest complexity are evaluated first. Mainly complex child nodes, so all nodes which are not implementing `ChoiceNode`, have to be evaluated last in process. This may accelerate the evaluation in some cases.

Figure 3.17 demonstrates an example for such an optimisation. In a first step the child nodes of the `and` are exchanged because the evaluation of the position node has a fixed complexity while for the evaluation of the vector norm a list of domains of definitions exists. Afterwards the `and` node and the index node are exchanged so that the most complex node is the last one to be evaluated.

Usage of Tautologies

The second approach is to use tautologies in order to minimise the effort needed for the evaluation of the operator tree. The underlying idea is that for a lot of expressions an equivalent expression exists whose evaluation requires less calculations and comparisons. In general, the fewer nodes are within the tree, the faster it can be evaluated.

The following formulas (3.10) to (3.16) contain some examples for possible optimisations which are easy to implement and can be detected fast since a complex

detection might outweigh the time obtained because of the rearrangement.

$$(\neg A \wedge \neg B) \Leftrightarrow \neg(A \vee B) \tag{3.10}$$
$$(\neg A \vee \neg B) \Leftrightarrow \neg(A \wedge B) \tag{3.11}$$
$$\neg(\neg A) \Leftrightarrow A \tag{3.12}$$
$$(A \vee A) \Leftrightarrow A \tag{3.13}$$
$$(A \wedge A) \Leftrightarrow A \tag{3.14}$$
$$(A \vee \neg A) \Leftrightarrow \top \tag{3.15}$$
$$(A \wedge \neg A) \Leftrightarrow \bot \tag{3.16}$$

Formulas (3.10) and (3.11) contain de Morgan's law[22, p. 60ff]. Here the expressions on the righthand side consist of one connective less than the ones on the lefthand side. Dealing with only two operands, the replacement because of de Morgan's law might slow down the evaluation because the binary connective is replaced. Since the evaluation of an `and` or `or` is stopped by most programming languages as soon as the result is unique, it might happen that the original variant requires less calculations. But de Morgan's law is also valid for the conjunction or disjunction of more than two propositions. In this case the optimisation speeds up the evaluation because the probability to abort the evaluation decreases with every new operand added to the expression. Figure 3.18 contains an example for an optimisation using de Morgan's law.

The lefthand sides of the formulas (3.12), (3.13) and (3.14) are expressions equivalent to a simpler one, namely a duplicate negation and a conjunction respectively disjunction of a proposition with itself. In figure 3.19 the case is displayed that $A \wedge A$ is replaced by A. Here the same letter within the choice node means that these nodes have the same type and parameters.

In formulas (3.15) and (3.16), an expression is a tautology or a contradiction, so always true or false. If this expression is part of another expression, the outer one can be simplified as follows:

In case of a conjunction, the result is false if one of the operands is a contradiction. If the operand is a tautology, it can be skipped because it does not influence the result. Dealing with a disjunction, a contradiction can be skipped whereas the result is always true if an operand is a tautology.

Figure 3.20 demonstrates how the use of formula (3.16) simplifies an expression. In a first step, $A \wedge \neg A$ is replaced by a contradiction. $\bot \vee B$ is equivalent to B so that the original complex tree in the end is equivalent to the simple evaluation of a single choice node.

To sum up, the operator trees can be optimised with the help of two different approaches. On the one hand, the order of child nodes can be exchanged so that choice nodes are evaluated before connective nodes and choice nodes with a lower

complexity are examined first. On the other hand, known tautologies can be used to minimise the number of nodes within the tree to obtain a faster evaluation of the full tree. In all cases it is important to deliberate out wether the effort of optimisation is smaller than the benefit. Here it has to be noticed that the optimisation happens once after parsing the operator file but the evaluation has to be done multiple times.

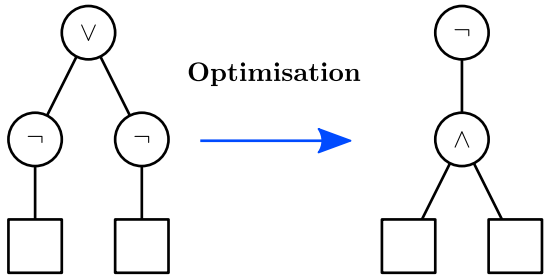


Figure 3.18: This optimisation uses de Morgan's law.

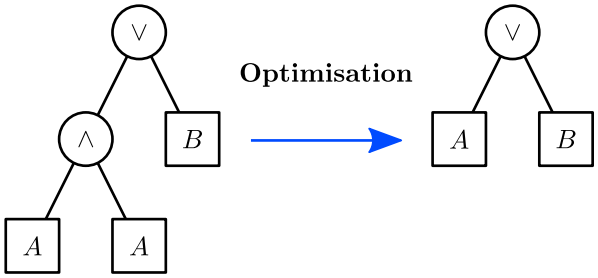


Figure 3.19: This optimisation uses $A \wedge A \Leftrightarrow A$.

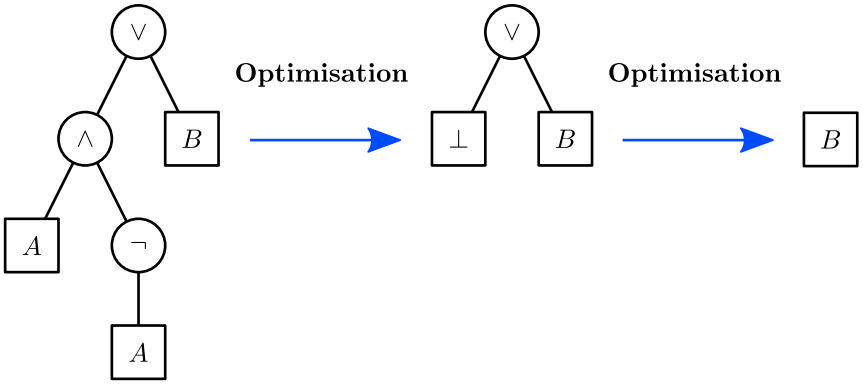


Figure 3.20: This optimisation uses $A \wedge \neg A \Leftrightarrow \bot$.

3.5 Output Definition via Filters

3.5.1 General Information

In the previous section is described how a subset is defined by an operator. As already mentioned in section 3.3, a view of the input data consists of an operator and a filter defining how the data subset is visualised. This section deals with the implementation of the filters.

There are mainly two possible kinds of filters. First, the particles can be directly drawn into a single three-dimensional scene. Second, they can be displayed as overlays being in the foreground of the three-dimensional scene. Moreover, there are different kinds of overlays: two-dimensional plots of the scene, two-dimensional diagrams and informational texts.

The filters describe how the particle data is to be displayed, they do not display them directly. This is important in order to encapsulate the visualisation framework. The "Visualisation" module (see figure 3.1, page 25) is responsible for the actual displaying part, whereas the "Views" module has got a descriptive character only.

3.5.2 Definition of Filters in Input Files

The views are to be defined by user written XML files. Therefore also filter files have to be enclosed since a view file only defines which operator is linked to which filter.

Some parameters have to be set for all kinds of filters like the colour or a unique identifier for the filter. Others fit only to scenic interpretations of the data like the particle shape or the shape size. This implies a derivation hierarchy between the different kinds of filter. XML files also offer the possibility of derived complex types which suits to the development of a filter hierarchy.

In the following examples it is demonstrated how filters can be defined in XML files. They do not cover all possibilities but are exemplarily for the majority of filters. Further explanations concerning all XML input files can be found in appendix B.

3D Scene

This filters (see listing 3.2) defines a three-dimensional scene which is labeled by the `<scene_3D>` tag. All inner tags of this tag define the particular visualisation parameters.

Within the tag `<color>`, the information about the particle colour is given. In this case the scalar property with the identifier `ChargeID` defines the colour. The tag `<attributeDomain>` defines that the expected property values are within $[0, 1]$. The colours are interpolated between white (`<from>1 1 1</from>`) to

red (<to>1 0 0</to>). The colours are defined in RGB[11] space. For more information about the colour spaces and interpolation see section 3.5.4.

The <size> tag defines that the scalar property with the ID MassID defines the particle size. Here the values are expected to be within $[-0.5, 0.5]$. In this case the lowest and greatest value of the domain of definition are included in the domain. To define excluded limits, the XML file has to be extended by further attributes for the particular tags.

The last tag <points> defines the shape of the particles. In this case they will be displayed as pixels. Other possibilities are <arrows3D> or <boxes> which might expect further parameters.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <filter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-
               instance filter.xsd"
5
6          id="Filter1"
7          name="ExampleFilter1"
8          description="Example for a filter">
9          <scene_3D>
10             <color ID="ChargeID">
11                 <attributeDomain>0 1</attributeDomain>
12                 <from>1 1 1</from>
13                 <to> 1 0 0 </to>
14             </color>
15             <size ID="MassID">
16                 <attributeDomain>-0.5 0.5</attributeDomain>
17             </size>
18             <points />
19         </scene_3D>
20 </filter>
```

Listing 3.2: Example of a 3D-Scene Filter

2D Scene

This filter (see listing 3.3) describes the interpretation of data in a two-dimensional scene which is indicated by the tag <scene_2D>. It can be applied to an overlay view. Colour and size are set analogously to the three-dimensional scene filter. Also the variants for the particle shape are defined in analogy to the three-dimensional variant with corresponding tags (see appendix B).

New with this filter is the <axes/> tag defining which of the three coordinates of the particle position has to be applied to which of the two-dimensional scene. In this case the first axis contains the x-coordinate, the second one gets the

[11] RGB stands for "Red, Green, Blue". This colour space defines all colours out of these three. Mostly, 256 different values can be applied to each of the colour components.[14, p. 15ff]

y-coordinate. Furthermore, it has to be defined how the third, until now unused, axis has to be handled. Possible are the two variants `missing=cuttingPlane`, with its attribute `cuttingPlane`, or `missing=all`. In the cutting plane variant, a domain of definition for the third axis has to be defined. All particles are displayed whose unused coordinates are within this domain, the others are dropped from the filter.

```
<filter>
    <scene_2D>
        <color ID="ChargeID">
            <attributeDomain>0 1</attributeDomain>
            <from>1 1 1</from>
            <to> 1 0 0 </to>
        </color>
        <size ID="MassID">
            <attributeDomain>-0.5 0.5</attributeDomain>
        </size>
        <points />
        <axes first="x" second="y" missing="cuttingPlane"
           cuttingPlane="0.3 0.7" />
    </scene_2D>
</filter>
```

Listing 3.3: Example of a 2D-Scene Filter

2D Diagram

The definition of a two-dimensional diagram has some similarities with the scene filters likewise (see listing 3.4). The colour of the graph is defined in the same way. The first difference compared to the scene filters is that the `<diagram_2D>` tag has got the attribute `diagramType` containing the graph type.

Both axes of the diagram have to be defined separately. For both, the ticksize and the label have to be defined. Concerning the data to be applied to the axes, a lot of possibilities exists. Each axis can contain the time or the average values of a physical or simulation property.

In the given example, the first axis contains the time in the mode `fixedInterval` which means that the axis contains a time interval of a fixed length. If a step has been loaded which is newer than the newest one displayed, the interval is moved so that the new step is included. The alternative is to define a given starting point for the interval so that it grows with each newly read step (`growingInterval`).

The second axis contains the average value of a scalar property. The diagram will contain the average value of the pressure of all chosen particles. If the operator has chosen only one particle, its pressure is displayed.

```
<filter>
    <diagram_2D diagramType="cross" >
        <color ID="MassID">
            <attributeDomain>0 1</attributeDomain>
            <from>1 1 1</from>
            <to> 1 0 0 </to>
        </color>
        <xAxis ticksize="0.05" label="Time">
            <time mode="fixedInterval" intervalLength="10" />
        </xAxis>
        <yAxis ticksize="0.5" label="Average measure" >
            <scalar ID="PressureID">
                <domain>0 0.75</domain>
            </scalar>
        </yAxis>
    </diagram_2D>
</filter>
```

Listing 3.4: Example of a 2D-Diagram Filter

3.5.3 Developed Class Structure

The previous section contains some examples for XML files defining filters. They have to be validated and parsed into an internal object structure to be used by the framework. Like with the operators, the validation can be done using the XSD file “filter.xsd”. While parsing a filter file, the information is stored using the class structure in figure 3.21.

As this diagram shows, each filter consists of a `Filter` object having an attribute of the type `FilterVisualisationAttributes`. The `Filter` itself contains all metadata like the identifier, the name, the description and the type of the filter. The `attributes` parameter contains all information for the display.

To decide how the particle data has to be visualised, it is important to know the filter type which is registered in the `type` object attribute. This parameter is automatically set by the constructor of each filter class. The filter provides the method `instanceof()` to check the type.

Interfaces for the Visualisation Attributes

As already mentioned, some of the displaying attributes like the colour have to be set for all kinds of filters. Others, like the particle size or the graph type of the diagrams, belong to only some of the filter types. The derivation hierarchy of the `FilterVisualisationAttributes` unifies common attributes as early as possible. This class contains only the parameters needed to define the particle, graph or text colour, namely the identifier of the property to be used (`colorID`), the domain of definition of this property (`colorDomain`) and two colours to be interpolated (`colorRange`).

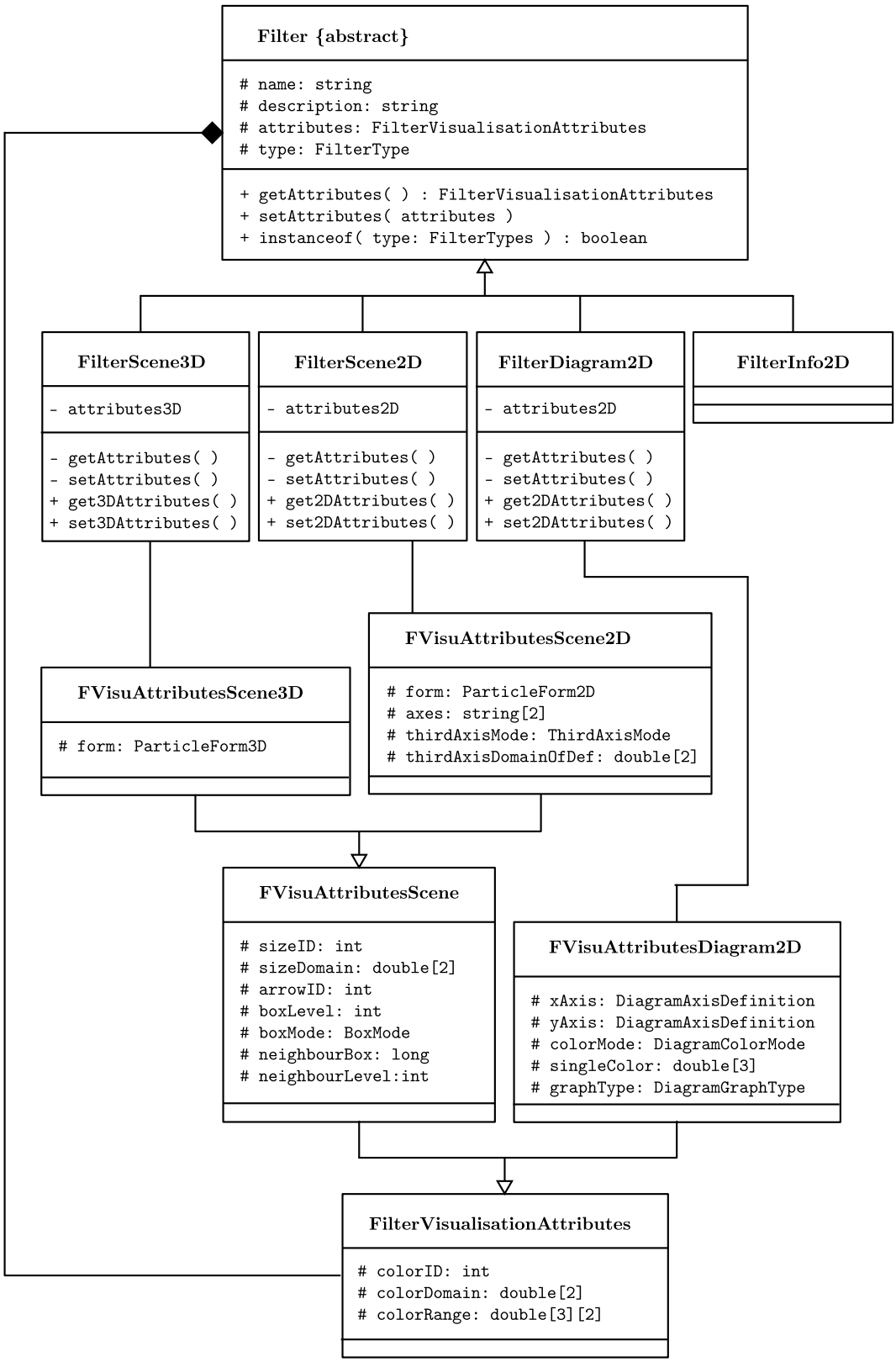


Figure 3.21: This UML class diagram contains the class structure used to store filters.

The class `FVisuAttributesScene` is directly derived from `FilterVisualisationAttributes` and extends its super class by further attributes needed for all scenic interpretations of the particle data.

Implementation of Visualisation Attributes

- The classes `FVisuAttributesScene3D` and `FVisuAttributesScene2D` are derived from the class `FVisuAttributesScene`. Both add the particle shape to the list of attributes, so wether the data has to be interpreted as pixels, boxes, spheres or arrows. In addition, the two-dimensional variant configures which coordinates of the position are mapped to the two axes and how the third unused axis has to be handled. The 3D-scene filter can be applied to a main view. All others can only be part of an overlay view.
- The class `FVisuAttributesDiagram2D` is directly derived from the base class `FilterVisualisationAttributes`. Both diagram axes have to be configured, i.e. which parameters or properties shall be used for the axes.
- The class `FilterInfo2D` defines which simulation parameters have to be displayed.

3.5.4 Colour Spaces and Interpolation

For all filters, the colour of the displayed figures like particles, boxes, graphs or texts is user-definable. In case of scene or diagram filters, it is possible to connect a particle property with the colour so that its distribution can be read out of the visualisation. Therefore additional information is needed, namely the identifier of the property to be used, the domain of definition of this property and two colours to interpolate between them.

For the interpolation, the first colour is applied to the lower bound of the domain of definition of the property, the second colour is applied to the upper bound. For all particles, the respective property value is used to interpolate between these colours.

To specify the colours, different colour spaces like RGB, HSB[12] or CMYK[13] can be used. For the interpolation in each colour space, there are a lot of possibilities. First, the colours can be interpolated linearly, discretely or with maximum contrast. The following example describes a way of linear interpolation in the RGB space.

[12] This colour space defines each colour using hue, saturation and the value brightness.[18, p. 60ff]

[13] CMYK stand for Cyan Magenta Yellow Key. It is a colour space mostly used for the printing process. The key defines the part of black within the colour, the other three values the parts of the corresponding colours.[21, p. 11f]

Example: Linear Interpolation in RGB Space

To interpolate between two RGB colours, the following formula for linear interpolation can be used. Let a_0 be the smallest property value, a_n the highest. So the domain of definition for the property values is $[a_0, a_n]$. The triple (r_0, g_0, b_0) is the colour used if the value is at the lower bound of this domain, (r_n, g_n, b_n) is the colour used at the upper bound of the domain. Now the colour for a particle with the attribute value a_i can be calculated by:

$$(r_i, g_i, b_i) = \left(r_0 + \frac{a_i - a_0}{a_n - a_0} \cdot (r_n - r_0), g_0 + \frac{a_i - a_0}{a_n - a_0} \cdot (g_n - g_0), b_0 + \frac{a_i - a_0}{a_n - a_0} \cdot (b_n - b_0)\right) \tag{3.17}$$

So all three colour components are linear interpolated independently of the others. Therefore the interpolation of each component is comparable with the interpolation between black and white. The following table 3.9 uses formula (3.17) to interpolate between black and white. Here the domain of definition of the property values, e.g. charge or pressure, is expected to be $[-3, 15]$.

Property a_i	-3	0	3	6	9	12	15
Colour	0	$0.1\overline{6}$	$0.\overline{3}$	0.5	$0.\overline{6}$	$0.8\overline{3}$	1

Table 3.9: This table demonstrates the linear interpolation of one colour component for discrete property values between black and white in RGB space.

3.6 Data storage

In the previous sections it has been described how the operators define subsets of the input data and how the graphical interpretation of these subsets is defined by filters. This leads to the question how the input data is stored by the framework. The question will be answered in this section.

3.6.1 Expected Input Data

The input data expected by the framework consists of particle data. It also consists of several time steps. For each time step, a particular amount of particles exists. The data of each particle contains the particle position, the values of particular physical properties (see section 2.1.3) and simulation properties like a particle index.

Not every particle has to exist in each time step since they can leave the simulated area. Therefore the number of particles as meta information can be explicitely given for each time step.

The input data can be read on different ways, e.g. out of data streams or from files stored on the local devices or in external storage devices. Thence it has to be stored in an internal structure to be used by the framework.

3.6.2 Data Structure

To develop an internal data structure for the storage of the read input data, some decisions have to be made, the first is how to sort the data. There are two possible approaches, namely to sort by the time steps or by the particles. The visualisation displays always subsets of a single time step. This leads to the conclusion that sorting by time steps is more suited for the application.

Storage of Particles

A second decision concerns the data structure within each time step. In this case there are three possibilities. The first one is to store the particle characteristics of all particles in a single multidimensional field or matrix. The disadvantage of this approach is that the different particle properties have various data types with different memory requirements. The field must consist of a data type which can include all other used ones so that this variant would result in a waste of memory.

A second possibility is to use different multidimensional fields or matrices, each one containing a particular data type. This way the characteristics of a single particle would be distributed to the fields. For the evaluation of an operator and for the visualisation, most times different properties of the same particle are needed. So if the data set is too large to fit in the small faster memory devices of the computer, the different fields would have to be reloaded again and again from the larger but

slower ones[14]. This decreases the performance of the entire framework.

The third approach is to create an object for each particle so that a particle class has to be defined. This way a time step consists of a list or field of particle objects. The use of objects is connected with an administration overhead but this approach has got two main advantages: First, all characteristics of a particle are united in a data type. Second, the value `NULL` can be used to define a non-existing particle. Therefore this third variant is used in the developed framework.

Storage of Time Steps

Dealing with large data sets, it is not reasonable to keep all time steps in the program storage during the runtime. The reasons are as follows. First, the user of the visualisation would have to wait a long time until all steps have been loaded, and all steps might be too large for the provided device. Moreover, this approach would presume that all time steps are already available. In case of online-visualisations, this demand is not fulfilled since chronologically newer steps are simulated while the older ones are already visualised. Therefore it is reasonable to store only a part of all time steps so that it has to be examined how to define this subset.

The most common usage of a visualisation is to switch from each time step to the chronologically following one. Sometimes the visualisation is stopped to examine a step in more detail or it is switched to the previously displayed step to compare the changes and trends. Therefore a temporal locality of the usage of input data can be supposed.

This leads to the idea to store a chronological set of time steps around the currently visualised one. This approach is illustrated in figure 3.22. A *Timesteps* object contains a circular array, so a queue, of *Timestep* objects. It uses pointers to determine which is the currently visualised step and which are chronologically older or newer.

Each *Timestep* has got an index to identify the step. Furthermore, it has got a list of all particles which exist in this step. A *Particle* object has got a unique identifying index, the particle position and lists of scalar and vectorial properties. For each data type, there are separate lists which are compressed to only two lists in this figure. A *Mapper* object defines which property is stored in which list at which position since this information is the same for all particles in all time steps.

The UML class diagram 3.23 demonstrates how the data structure is realised in the developed framework. It shows that four different data types are supported, namely `integer`, `float`, `double` and `long`.

This storage structure makes it possible to store only those particles which are chosen by at least one of the loaded operators. This way the amount of time steps which have to be always stored by the framework during runtime can be increased. In general, the user has to define how many time steps shall always be stored since

[14] In the following enumeration, each storage device is slower than the preceding one but offers more memory: cache, main storage, hard drive, external storage.

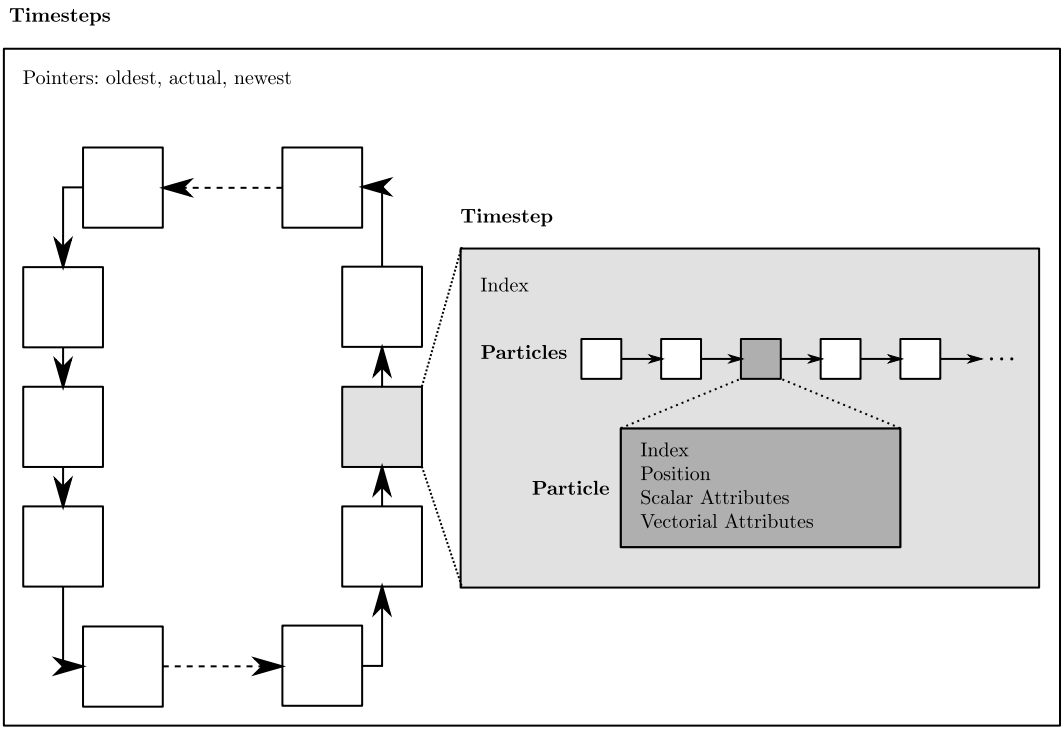


Figure 3.22: A *Timesteps* object contains a chronological set of *Timestep* objects around the currently visualised one. A *Timestep* has got a list of *Particle* objects.

the optimal amount depends on the memory requirements of each time step and the size of the storage devices of the computer used for the visualisation.

If always three or more steps are visualised, so the currently displayed, the previous and the following one, it is possible to preload the steps most probably needed next in a separate thread to increase the performance.

Conclusion

The framework uses a data structure which sorts the input data by the index of time steps. The characteristics of each particle are stored so that each particle is a separate object. To benefit from the temporal locality of a visualisation, only a chronological subset of time steps around the currently visualised one is stored by the program. A time step contains a list of all particles within this step. Each particle object administrates the different characteristics sorted by the dimensionality of their data vector and by their data types.

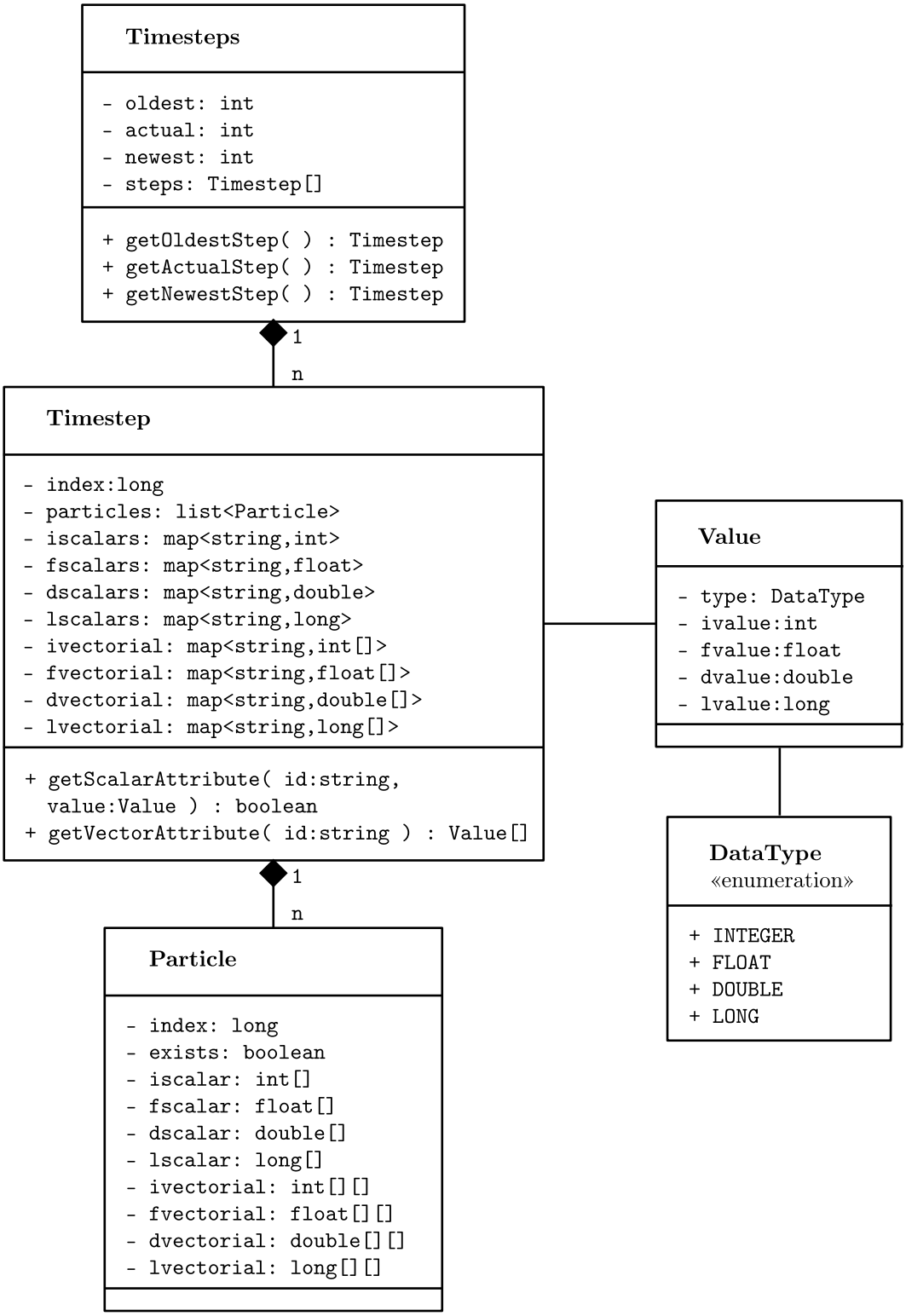


Figure 3.23: This UML class diagram illustrates how the data storage is realised within the framework.

4 Conclusion

In this thesis the development of a "Multiple View Visualisation Framework for Large Particle Sets" is described. At first, already existing frameworks, namely ParaView and VisIT, have been evaluated: Both provide an enormous amount of features. Due to this it is difficult to find the needed features in the user interfaces. Furthermore, these frameworks are not specialised for particle data and thus not optimal for large data sets of this type. This led to the demand for the development of another visualisation framework specialised for particle data.

For the development of this framework, the most interesting and best-fitting features concerning large particle data sets have been combined. These are in detail the remote output and the concepts of operators and filters. An operator is used to define a subset of the given input data, a filter defines how such a subset is graphically interpreted. Both ideas have been unified in the concept of multiple views.

The concept of multiple views constitutes the main aim of the developed framework. Each of the views consists of an operator and a filter. The operators are defined using the tree structure of XML files so that they are (theoretically) unlimited complex and reusable. The filters are also defined in XML files and specialised for particle data sets. They provide for example two- and three-dimensional scenes, diagrams and informal textual output.

The new approach unifies all three-dimensional scenes in the same window providing additional views with other filter types as overlays to it. Other frameworks also provide different multiple views but only in sub- or different windows. This approach led to further new aspects of the framework, namely multiple input and output.

The framework has been designed modularly to provide extensibility. Data input streams are readable as well as classical file input. The visualisation is displayable on the screens of desktop computers as well as on stereoscopic projection systems like the one installed in the Rotunde at Jülich Supercomputing Centre. The idea of multiple views in the same window suits the different outputs since its appearance is adequate for the different output media. To give an example, several windows on stereoscopic systems weaken the impression of three-dimensionality. To provide this feature of overlay views and also for the remote visualisation, the toolkit ViSTA has been used.

A second aspect is the data storage module adapted to particle data sets with potentially thousands of time steps each containing up to billions of particles. This module stores only a chronological set of time steps around the currently visualised one. The following steps are automatically preloaded by a loader module which increases the performance of the entire framework.

To sum up, the developed framework provides four approaches, namely multiple input and multiple output, the concept of multiple views in a single window as well as a data storage concept adapted to large particle data sets.

5 Outlook

During the development of the framework, some aspects have become apparent for which the implementation shows bottlenecks so that there is still potential to improve them.

First, the adjustment of overlays is not always optimal in case that all borders are nearly completely filled. It might not be possible to add another one since there is no border with enough continuous free space. A reordering of the already added overlays might result in a gap large enough to fit in the new overlay. Therefore it is reasonable to improve the adjustment algorithm.

Another aspect concerns the display of diagrams as overlays. Using ViSTA, it is impossible to use diagrams created with already existing plot libraries since for example the OpenGL commands defining the diagram have to be included directly into ViSTA classes. Hence, the display of diagrams is possible but it has to be done without a plot library. This results in a large effort for the implementation.

The last aspect to improve the framework is related to the data storage. Here some methods return objects instead of pointers or references. This way, for each function call, the copy constructor of the object has to be called. Returning a pointer instead would increase the performance of the framework. This bottleneck can be avoided with an acceptable effort.

So to sum up, there are three possible approaches to improve the framework. They are all realisable and do not require larger changes in the framework due to its modular structure.

To outline the future direction of the framework, some features are mentioned in the following which are desired but have not yet been implemented.

GUI Implementations

Until now, a module is provided which serves as a communication port for different GUI implementations though no GUI has been implemented yet. Since the framework is designed to use multiple output devices, also the user interfaces required by the several use cases differ a lot. If the screen of a desktop computer or a laptop is used as output device, the GUI runs on the same computer displayed in a separate window.

In case of a visualisation on a stereoscopic projection system, possibly with optical tracking, a GUI displayed at the same projection system would weaken the impression of three-dimensionality. Therefore, an interface running on tablet computers or even a smartphone should be implemented.

Both types of user interfaces require different designs since the small displays of tablet computers or smartphones offer only limited space whereas for desktop computers larger monitors are installed. Therefore different GUI implementations are needed. Since the visualisation framework offers a communication module us-

ing TCP/IP and a message protocol, the only demand on the GUIs is to use this protocol and to send the messages to the correct IP and port.

Different Data Formats and Input Modes

The framework has been designed to provide multiple input. This means that on the one hand different modes or ways of input like streams and files on local or external storages are supported. On the other hand, the data format within the stream or file is not fixed. Until now, only the use of a separate file for each time step has been implemented. This is extensible so that all time steps in the same file are readable. This mode can already be defined in the configuration XML files (see appendix B). Further modes like reading the input out of streams have not yet been adapted to the framework but the corresponding modules can be easily appended by this feature. Also a direct access to SIONlib[1] would be reasonable.

Fast Particle Prefiltering

In the current implementation, all existing particles are stored within the time step objects. In case of huge amounts of particles within each step, it is reasonable to store only those particles which are chosen by at least one of the operators. This particle prefiltering might speed up the performance of the entire framework.

If the union of all choices contains nearly all particles, the effect is small but mainly in case of particle tracking the amount of data to be stored would be much smaller this way. Hence more time steps can be stored by the framework with the same memory consumption, the visualisation runs smoothly.

Parallel Input

In the last paragraphs it has been described that the different time steps can be set up in a separate file for each time step. Mainly in this case it is possible to implement a parallel input to speed up the loading progress. There are several approaches how to parallelise the loading procedure. To give an example, the easiest one is to load each step in a new thread with several loading threads. Since already each step is loaded by a newly created thread, the only change would be to start more than one loading thread at a time.

To sum up, there are a couple of approaches to implement further features for the visualisation framework. Some of them are based on the aspect to speed up the performance, others deal with user-friendliness or ways to offer the framework for a larger group of users since each simulation has its own output formats which have to be recognized by the framework.

[1] SIONlib stands for “Scalable I/O library” and can be used “for parallel access to task-local files”[10]. It is developed at Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH.

Bibliography

[1] Virtual Reality. In *The American Heritage Science Dictionary*. Houghton Mifflin Company. Available online at `http://dictionary.reference.com/browse/virtualreality`; visited on June 21st 2011.

[2] 3Dconnexion. Spacenavigator – Die 3D-Maus. Available online at `http://www.3dconnexion.de/products/spacenavigator.html`; visited on July 7th 2011.

[3] A. Andres. Visualisierung. In H.-J. Schneider, editor, *Lexikon Informatik und Datenverarbeitung*. R. Oldenburg Verlag, 1997.

[4] T. Beer. VR Software ViSTA. Available online at `http://www.rz.rwth-aachen.de/aw/cms/rz/Themen/Virtuelle_Realitaet/research/~piz/vr_software_vista/?lang=de`; visited on June 21st 2011.

[5] R. L. Bisplinghoff, J. W. Mar, and T. H. H. Pian. *Statics of Deformable Solids*. Addison-Wesley (republished by Dover Publications (1990)), 1965.

[6] H. Dachsel. Fast Multipole Method. Available online at `http://www.fz-juelich.de/jsc/fmm/`; visited on June 6th 2011.

[7] H. Dachsel. An error-controlled fast multipole method. *The Journal of Chemical Physics*, 132, 11, 119901, 2010.

[8] J. M. Dawson. Particle simulation of plasmas. *Reviews of Modern Physics*, 55:403–447, 1983.

[9] K. Wu et al. Kinetics of water filling the hydrophobic channels of narrow carbon nanotubes studied by molecular dynamics simulations. *The Journal of Chemical Physics*, 133, 2010.

[10] Wolfgang Frings. SIONlib. Available online at `http://www2.fz-juelich.de/jsc/sionlib/`; visited on June 29th 2011.

[11] M. Griebel, S. Knapek, G. Zumbusch, and A. Caglar. *Numerische Simulation in der Molekulardynamik – Numerik, Algorithmen, Parallelisierung, Anwendungen*. Springer-Verlag, 2004.

[12] Gyration. Homepage of Gyration. Available online at `http://www.gyration.com/index.php/us/home.html`; visited on July 7th 2011.

[13] K. Hawkins and D. Astle. *OpenGL Game Programming*. Prima Tech, 2001.

[14] Keith Jack. *Digital Video and DSP: Instant Access*. Newnes, 2008.

[15] R. S. Wright Junior and M. Sweet. *OpenGL SuperBible*. Waite Group Press, 2nd edition, 2002.

[16] I. Kabadshow and H. Dachsel. The Error-Controlled Fast Multipole Method for Open and Periodic Boundary Conditions. In *Fast Methods for Long-Range Interactions in Complex Systems*. ISBN: 978-3-89336-714-6, also available online at `http://www2.fz-juelich.de/wehss/lecture_notes/`; visited on July 8th 2011.

[17] Kitware. ParaView. Available online at `http://paraview.org`; visited on June 21st 2011.

[18] Andreas Koschan. *Digital Color Image Processing*. Wiley-Interscience, 1st edition, 2008.

[19] Lawrence Livermore National Laboratory. VisIT. Available online at `https://wci.llnl.gov/codes/visit/home.html`; visited on June 21st 2011.

[20] B. Lahres and G. Raýmon. *Objektorientierte Programmmierung – Das umfassende Handbuch*. Galileo Computing, 2009.

[21] S. Nagabhushana. *Computer Vision And Image Processing*. New Age International Pvt Ltd Publishers, 2009.

[22] O. Opitz. *Mathematik – Lehrbuch für Ökonomen*. Oldenbourg, 1997.

[23] G. Saake and K.-U. Sattler. *Algorithmen und Datenstrukturen – Eine Einführung mit Java*. dpunkt.verlag, 2006.

[24] American Meteorological Society. Glossary of Meteorology. Available online at `http://amsglossary.allenpress.com/glossary/search?id=particle1`; visited on June 6th 2011.

[25] V. Springel et al. Simulating the joint evolution of quasars, galaxies and their large-scale distribution. *Nature*, 435:629–636, 2005.

[26] S. Y. Trofimov, E. L. F. Nies, and M. A. J. Michels. Constant-pressure simulations with dissipative particle dynamics. *The Journal of Chemical Physics*, 123, 2005.

[27] T. van Reimersdahl, T. Kuhlen, A. Gerndt, J. Henrichs, and C. Bischof. ViSTA: a multimodal, platform-independent VR-Toolkit based on WTK, VTK, and MPI. In *Fourth International Immersive Projection Technology Workshop (IPT2000), Ames, Iowa*, 2000. Available online at `http://www.rz.rwth-aachen.de/aw/cms/rz/Themen/Virtuelle_`

Realitaet/research/~piz/vr_software_vista/?lang=de; visited on June 21st 2011.

A Dependency Graph of the Modules

This graph displays the main dependencies between the components of the different modules. Not every class and interface is illustrated but only the main ones. The classes `Administration`, `Timesteps` and `AttributeMapper` appear in both parts of the dependency graph.

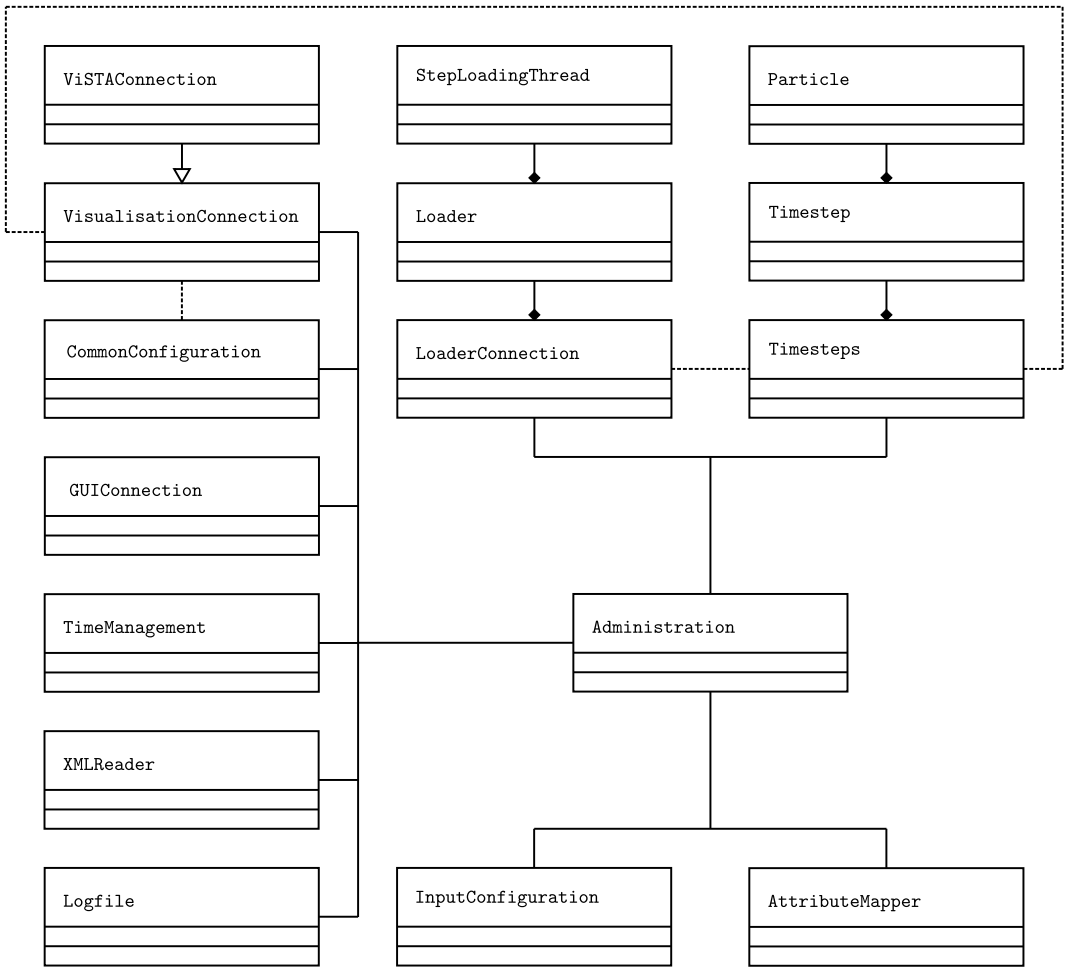


Figure A.1: This is the first part of the dependency graph of the different module components.

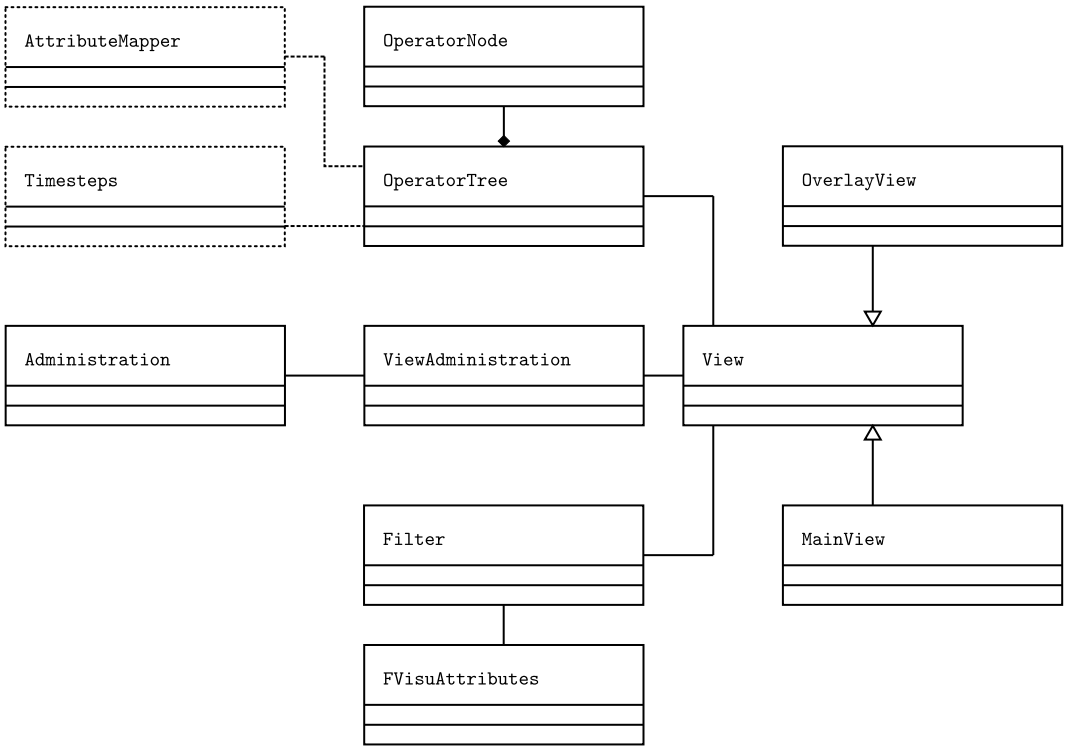


Figure A.2: This is the second part of the dependency graph of the different module components. The classes `Timesteps` and `AttributeMapper` have been introduced in part one.

B Example XML Files

B.1 Input Configuration

Since the format of the input files differs a lot, it has to be defined how the data is stored so that the loader module is able to read in the particle data. This configuration is done in the input configuration. The following file is an example for such an XML file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-
               instance input.xsd" >
5      ❶<common path="/home/user/sim/data/"
6              ending="dat"
7              characterSet="UTF-8"
8              title="Simulated Data"
9              description="An optional description." >
10         <separator values=" ">
11             <dataSetSeparator>\n</dataSetSeparator>
12         </separator>
13         <files name="xe9093_full" separatorNamePath="-">
14             <numbering startIndex="1" maxIndex="500"
                   leadingZeros="false" />
15         </files>
16     </common>
17
18     ❷<data numberOfParticles="1000000">
19         ❸<index exists="false"/>
20         ❹<position type="float">
21             <coordinate axis="x" column="1" />
22             <coordinate axis="y" column="2" />
23             <coordinate axis="z" column="3" />
24         </position>
25         ❺<attributes>
26             ❻<scalar ID="C"
27                     name="Charge"
28                     type="double"
29                     use="true"
30                     column="4" />
31             ❼<vector ID="V"
32                     name="Velocity"
33                     type="float"
34                     use="true" >
35                 <coordinate axis="x" column="5" />
36                 <coordinate axis="y" column="6" />
37                 <coordinate axis="z" column="7" />
38             </vector>
```

```
39              ❽<timestepScalar ID="TM"
40                               name="Total Mass"
41                               type="long"
42                               use="true"
43                               labelBefore="TOTAL_MASS="
44                               labelAfter="\n"
45                               position="top" />
46              ❾<timestepVector ID="Ctrd"
47                               name="Centroid"
48                               type="float"
49                               use="true"
50                               labelBefore="CENTRE="
51                               labelAfter="\n"
52                               position="bottom" >
53                  <coordinate axis="x" column="1" />
54                  <coordinate axis="y" column="2" />
55                  <coordinate axis="z" column="3" />
56              </timestepVector>
57          </attributes>
58      </data>
59  </config>
```

Listing B.1: Example for an Input Configuration XML File

The configuration XML file consists of two main parts, the `<common>` part giving file meta information and the `data` part defining the inner file structure.

❶ The `<common>` part defines the meta information about the input file(s). The `<separator>` tag defines how the data sets are separated, i.e. the particles and the values within the data set. The existence of the `<files>` tag determines that a separate file exists for each time step.

❷ The `<data>` tag defines which particle properties are stored with what inner structure. For the following description an inner structure with lines and columns is requested.

❸ The `<index>` tag determines if a particle index exists and in which column it is stored.

❹ This tag keeps the information in which columns the coordinates of the particle position can be found.

❺ The `<attributes>` tag contains information about all particle properties. In this case a scalar and a vectorial property exist for each particle (❻, ❼). Furthermore, a scalar and a vectorial property exist belonging to the entire time steps information (❽, ❾).

B.2 Further Configurations

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-
               instance common.xsd" >
5      ❶<visualisationAttributes>
6          <startPosition>0.5  0.5  2.0</startPosition>
7          <viewDirection>0.0  0.0  1.0</viewDirection>
8          <backgroundColor>0.0  0.0  0.0</backgroundColor>
9      </visualisationAttributes>
10     ❷<connection host="localhost"
11                  port="2111" />
12     ❸<other  velocity="500"
13             stepsToBeLoaded="3"
14             logfile="Logfile.txt" />
15 </config>
```

Listing B.2: Example of an XML File for Further Configurations

This XML configuration file contains further configurations which belong mainly to common visualisation aspects.

Tag ❶ defines a couple of general visualisation parameters for the start of runtime. There are the start position and viewing direction within the three-dimensional scene as well as the background colour.

With tag ❷, the IP and port for the communication between the framework and a GUI are defined.

Tag ❸ defines additional parameters. The first is the length of the time lag between the display of two time steps in milliseconds (`velocity`). The parameter `stepsToBeLoaded` defines how many time steps are stored by the framework at the same time. The `logfile` parameter includes the file used as a log.

B.3 Views

The following examples demonstrate how a view is defined. The parameters to be given are a unique identifier, a name, an optional description, the operator and filter files to be used.

The existence of either the `<mainView>` or the `<overlayView>` tag defines the type of the view.

Main Views

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <view   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-
               instance view.xsd" >
5      <configuration  ID="MainView"
6              name="Main View"
7              description="This is a main view."
8              operator="operator2.xml"
9              filter="filter2.xml">
10         <mainView />
11     </configuration>
12 </view>
```

Listing B.3: Example of a Main View XML File

Overlay Views

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <view   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-
               instance view.xsd" >
5      <configuration  ID="2D_Overlay"
6              name="2D-Scene Overlay"
7              description="This is a two-dimensional scene
                  overlay."
8              operator="operator1.xml"
9              filter="filter1.xml">
10         <overlayView    border="right"
11                         order="1"
12                         width="300"
13                         height="300" />
14     </configuration>
15 </view>
```

Listing B.4: Example of an Overlay View XML File

B.4 Operators

The following examples are the simplest operators which can be defined. They contain a single choice node but no connectives. Section 3.4 also describes a more complex example.

Index

This operator chooses the subset with respect to the particle index. The first line contains information about the XML version and the file encoding. In the second and third line the XSD file "operator.xsd" and some further information are determined which are necessary to validate the XML file in general and against the schema definition. These three lines are skipped in all following operators since they are all the same.

Lines four to six contain the identifier, the name and a description of the operator. The ID has to be unique within all operators used in a run. The name is arbitrary, the description optional. Also the `<operator>` tag is omitted in all following examples.

The tag `<index>` shows that the subset has to be chosen using the particle index. This operator chooses all particles with the index between one and ten, the index 15 or indices between 100 and 1000.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-
       instance"
3             xsi:schemaLocation="http://www.w3.org/2001/
                  XMLSchema/instance operator.xsd"
4             id="Index"
5             name="IndexOperator"
6             description="Selection Criteria: Index" >
7      <index>
8          <domain>1 10</domain>
9          <domain>15</domain>
10         <domain>100 1000</domain>
11     </index>
12 </operator>
```

Listing B.5: Example of an Index Operator

Position: Cuboid

This operator uses the position of the particles to define the subset. All particles within a cuboid are chosen. The tag `<position>` defines that the particle position has to be used. The inner tag `<cuboid>` defines that a cuboid is the selection criterion. This cuboid has a range on the x-axis from 0.2 to 0.8, on the y-axis from 0.4 to 0.6 and covers the whole z-axis which is determined by the tag `<z_all />`.

```
<position>
    <cuboid>
        <x>0.2 0.8</x>
        <y>0.4 0.6</y>
        <z_all />
    </cuboid>
</position>
```

Listing B.6: Example of a Position Operator (Cuboid)

Position: Domains of Definition

This operator defines the subset as well using the position. It uses domains of definition for all three axes which are marked by the tag `<domain>`. Here the y-coordinate of the position does not influence the choice because of `<y_all />`. Only particles whose z-coordinate is within 0.3 and 0.7 and whose x-coordinate is either between 0.2 and 0.4 or between 0.6 and 0.8 are chosen.

```
<position>
    <domain>
        <x>0.2 0.4</x>
        <x>0.6 0.8</x>
        <y_all />
        <z>0.3 0.7</z>
    </domain>
</position>
```

Listing B.7: Example of a Position Operator (Domains of Definition)

Position: Sphere

This operator considers the particle position choosing all those particles within a sphere represented by the tag `<sphere>`. This shape has got the radius 0.4 and the centre $(0.1, 0.2, 0.3)$.

```
<position>
    <sphere radius="0.4" >
        <centre>0.1 0.2 0.3</centre>
    </sphere>
</position>
```

Listing B.8: Example of a Position Operator (Sphere)

Scalar Property

The tag `<scalar>` declares that this operator uses the scalar property with the identifier “MassID” to determine the subset. All particles weighing between 0.01 and 0.05 or exactly 1.0 weight units are chosen.

```
<scalar ID="MassID" >
    <domain>0.01 0.05</domain>
    <domain>1.0</domain>
</scalar>
```

Listing B.9: Example of a Scalar Property Operator

Vectorial Property: Coordinates

Because of the tag `<vector>`, the vectorial property with identifier "VelocityID" is used. The tag `<coord>` determines that all coordinates have to be evaluated individually. Because of `<x_all />`, the x-coordinate of the property value is not taken into account. The y-coordinate has to be within 0.0 and 0.5, the z-coordinate between 0.0 and 0.9 in order to choose the particle.

```
<vector ID="VelocityID" >
    <coord>
        <x_all />
        <y>0.0 0.5</y>
        <z>0.0 0.9</z>
    </coord>
</vector>
```

Listing B.10: Example of a Vectorial Property Operator (Coordinates)

Vectorial Property: Norm

This operator also uses the vectorial property with identifier "VelocityID", but the norm of the vector instead of the particular coordinates which is defined by the tag `<norm>`. The attribute `method="euklid"` determines that the Euclidian norm has to be used. All particles whose velocity is between 0.1 and 0.4 or at 0.5 units are chosen.

```
<vector ID="VelocityID" >
    <norm method="euklid" >
        <domain>0.1 0.4</domain>
        <domain>0.5</domain>
    </norm>
</vector>
```

Listing B.11: Example of a Vectorial Property Operator (Norm)

B.5 Filters

Three-Dimensional Scene

This filter defines a three-dimensional scene which is marked by the <scene_3D> tag. For all of these scenes, the colour and size of the displayed particle representations have to be defined. For both, the identifier of the property has to be defined as well as the domain of definition of the property values. For the colour, also the two colours for the interpolation have to be defined.

```
<?xml version="1.0" encoding="UTF-8"?>

<filter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-
        instance filter.xsd"
    id="Filter1"
    name="ExampleFilter1"
    description="Example for a filter">
    <scene_3D>
        <color ID="ChargeID">
            <attributeDomain>0 1</attributeDomain>
            <from>1 1 1</from>
            <to> 1 0 0 </to>
        </color>
        <size ID="MassID">
            <attributeDomain>-0.5 0.5</attributeDomain>
        </size>
        ❶<points />
    </scene_3D>
</filter>
```

Listing B.12: Example of a 3D-Scene Filter

Tag ❶ defines how the particle data is graphically interpreted. In the given example, the data is interpreted as points. The following alternatives exist:

```
❷<spheres />
❸<arrows3D ID="VelocityID" />
❹<boxes level="4" mode="all" />
❺<boxes level="4" mode="neighbours" neighbourBox="20"
   neighbourLevel="2" />
```

Listing B.13: Alternative Display Types (3D-Scene)

Possibility ❷ visualises spheres instead of points. In both cases it has to be noticed that particles are zero-dimensional but a property – in this case “MassID” – is adapted to the point size respectively sphere radius.

Variant ❸ adds an arrow to the displayed point so that a vector field can be displayed. Therefore the ID of a vectorial property has to be given so that the corresponding values are used for the arrow vector orientation.

The possibilities ❹ and ❺ do not display any particle but the boxes used by the FMM. In ❹ all boxes containing at least one particle are visualised, in ❺ only the chosen box and all filled neighbour boxes are displayed.

Two-Dimensional Scene

This filter defines a two-dimensional scene analogue to the previous three-dimensional one.

```
<filter>
    <scene_2D>
        <color ID="ChargeID">
            <attributeDomain>0 1</attributeDomain>
            <from>1 1 1</from>
            <to> 1 0 0 </to>
        </color>
        <size ID="MassID">
            <attributeDomain>-0.5 0.5</attributeDomain>
        </size>
        ❶<points />
        ❷<axes first="x" second="y" missing="cuttingPlane"
           cuttingPlane="0.3 0.7" />
    </scene_2D>
</filter>
```

Listing B.14: Example of a 2D-Scene Filter

For tag ❶, further variants are the same as for the three-dimensional case. The only difference is that instead of `<arrows3D>` the tag is named `<arrows2D>`.

Tag ❷ does not exist in three-dimensional scenes. It defines which coordinates of the particle position are adapted to which of the scene axes. Furthermore, it has to be defined how the third coordinate is handled. In the given example, all particles whose third coordinate is within the given range will be displayed. The alternative is to ignore the third coordinate completely so that all particles are displayed which have been chosen by the related operator:

```
<axes first="x" second="y" missing="all" />
```

Listing B.15: Alternative Axis Definition (2D-Scene)

Two-Dimensional Diagram

This filter defines a two-dimensional diagram. The first parameter to be adjusted is the diagram type ❶. Valid values are “point”, “cross”, “lines”, “interpolated_lines” and “bars”. The colour is defined analogue to the scene filters.

```
<filter>
    <diagram_2D ❶diagramType="cross" >
        <color ID="MassID">
            <attributeDomain>0 1</attributeDomain>
            <from>1 1 1</from>
            <to> 1 0 0 </to>
        </color>
        ❷<xAxis ticksize="0.05" label="Time">
          ❸<time mode="fixedInterval" intervalLength="10"/>
        </xAxis>
        ❹<yAxis ticksize="0.5" label="Average measure" >
            ❺<scalar ID="ChargeID">
                <domain>0 0.75</domain>
            </scalar>
        </yAxis>
    </diagram_2D>
</filter>
```

Listing B.16: Example of a 2D-Diagram Filter

With the tags ❷ and ❹ the axes of the diagram are defined. There are several possibilities which can be added to both of the axes. Variant ❸ defines to display the time in form of an interval with a fixed length. Tag ❺ determines to use the average value of a scalar property. Alternatives are the following ones:

```
❻<position>
    <coordinate coord="z" domain="0.0 1.0"/>
</position>
❼<position>
    <distance normMethod="euklid">
        <distanceRefPoint>0.5 0.5 0.5</distanceRefPoint>
    </distance>
</position>
❽<vector ID="CentreOfGravityID" >
    <coordinate coord="x" domain="0.5 0.8" />
</vector>
❾<timestep>
    <vector ID="CentreOfGravityID" >
        <norm normMethod="norm_max">
            <domain>0 2</domain>
        </norm>
    </vector>
</timestep>
```

Listing B.17: Alternative Axis Definitions (2D-Diagram)

❻ ❼ The average position, so the centre of mass, of the particles is used. It can be evaluated either using a particular coordinate or using the distance to a given reference point.

❽ The average of a vectorial property is used. Either the norm of this vector or a single coordinate are examined. The norm variant can be found in the inner tag of ❾.

❾ Here a time step instead of a particle property is used. This means that the value exists only once for the entire time. The inner tag can be of the types ❺, ❽ or the one used in this example.

C Example Input Files

The listing is an excerpt of an input file containing a single time step. Its structure is described by the input configuration XML file described in appendix B.1.

```
1  TOTAL_MASS=13524
2  .500000482697E+00 .499999411660E+00 .499948158408E+00 -.1E+01
       0.01 0.05 0.05
3  .500000232998E+00 .499999892997E+00 .500006356409E+00 -.2E+01
       0.02 0.04 0.03
4  .500000016695E+00 .499999645439E+00 .499898832246E+00 +.1E+01
       -0.01 0.03 0.08
5  .500036268437E+00 .499998883678E+00 .499975850994E+00 -.1E+01
       0.03 -0.03 0.00
6  [...]
7  .500011098384E+00 .499966061604E+00 .499976418459E+00 -.2E+01
       -0.01 -0.03 0.06
8  .499971883576E+00 .499978255637E+00 .499975357310E+00 +.1E+01
       0.01 0.04 -0.06
9  .499972410840E+00 .500020580326E+00 .499975916630E+00 -.2E+01
       0.06 -0.03 -0.01
10 .500010945213E+00 .500032875804E+00 .499976124376E+00 -.1E+01
       0.00 -0.01 0.00
11 CENTRE=0.512 0.4992 0.5013
```

Listing C.1: Example of an Input File

Line one beginning with "TOTAL_MASS=" belongs to tag ❽ of the input file. Line eleven starting with "CENTRE=" to tag ❾. These two lines contain the two properties which refer to the whole time step.

Each of the lines two to ten defines the characteristics of a particle for the given time step. The values in the first columns contain the particle position. The fourth value is the respective charge. The last three values define the velocity.

D Requirements

For the installation of the developed framework, the following toolkits and libraries are needed. Note that the specified versions are the ones installed for the test environment.

Library/ Toolkit/ Framework	Version	Available
C++ compiler (g++)	4.3.2	http://gcc.gnu.org/releases.html (GNU Compiler Collection, others also possible)
cmake	2.6	http://www.cmake.org/
Qt	4.7.0	http://qt.nokia.com/products/
OpenGL	1.3	http://www.opengl.org/sdk/
GLUT API	4	http://www.opengl.org/resources/libraries/
OpenSG	special	For this special version, contact ViSTA development team for the distribution[1]
ViSTA	1.10.1	http://sourceforge.net/projects/vistavrtoolkit/

Table D.1: Required Libraries and Toolkits

[1]For additional contact information see http://www.rz.rwth-aachen.de/aw/cms/rz/Themen/Virtuelle_Realitaet/research/~piz/vr_software_vista/?lang=de

E Tested Platforms

Desktop System

Operating System	Linux Workstation with openSUSE 11.1 (i686)
Compiler	GNU Compiler Collection (gcc), version 4.3.2
cmake	Version 2.6 patch 2
OpenGL	Version 1.3
GLUT	freeglut version 1, *GLUT* API 4
Qt	Version 4.7.0
OpenSG	special, see appendix D
ViSTA	Version 1.10.1

Stereoscopic Projection System

Operating System	Linux System with openSUSE 10.2 (i586)
Compiler	GNU Compiler Collection (gcc), version 4.1.2
cmake	Version 2.8.0
OpenGL	Version 1.1
GLUT	freeglut version 1, *GLUT* API 4
Qt	Version 4.7.3
OpenSG	special, see appendix D
ViSTA	Version 1.10.1

F License

The developed framework is available under the *GNU Lesser General Public License* which contains additional permissions to the *GNU General Public License*. The exakt definitions of these licenses can be found online:

http://www.gnu.org/licenses/lgpl.txt
http://www.gnu.org/licenses/gpl.txt

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.
The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:
a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.
You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:
a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
d) Do one of the following: 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.
e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.
You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by

this License, and convey such a combined library under terms of your choice, if you do both of the following:
a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Jül-4345
November 2011
ISSN 0944-2952